JOB EXECUTION IN A DISTRIBUTED ENVIRONMENT USING PETRI NETS

by

LING-LING HSU

B.S., Fu-Jen University, 1982

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

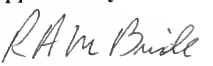
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:



Major Professor

Acknowledgements

I wish to thank my major professor, Dr. Rich A. McBride, for his guidance and encouragement during the process of this report. I would also like to thank my husband for reviewing this report.

Table of Contents

List of Figures

CHAPTER 1
INTRODUCTION

1.1 Overview

In the broadest sense, all resource-sharing systems in which two or more processors are involved can be described as distributed systems. A distributed system is characterized by having processors and storage facilities which are physically separated from each other. Also, data communication facilities are used to interconnect computers that are dispersed and play a more significant role than in a centralized or nondistributed system.

A distributed system performs the following three major activities [ENCY83]:

1) executes instruction sequences representing functions,

2) stores data at different locations,

3) allows control to be distributed among many sites.

A system which supports all these activities is said to exhibit a high degree of distribution. Functions, which can be treated as operators or modules, may exist at different sites. Data information can either be stored at only one site or at many different sites. Finally, distributed system control means that decisions can be made independently by one of the sites.

Petri nets [PETR76] provide a tool for modeling systems and the use of Petri nets for this purpose offers users several potential advantages. The overall system being modeled is often easier to understand due to the graphical and precise nature of the

representation scheme for Petri nets. Furthermore, the behavior of a system that is modeled can be analyzed using Petri net theory.

This report describes a version of the Petri net model which incorporates extensions for specifying and controlling the execution of tasks in a distributed environment. This extended Petri net can be used to provide a job's specification, which will be referred to as a Control net (CN). Moreover, transitions in a CN represent actions for a job to perform. Predicates are used in conjunction with transitions in the Petri net in order to control when these transitions may be performed.

A message which carries a CN is referred to as an intelligent token. This type of message is designed to carry control and data information which are needed to perform a task. A process called a Control net agent exists at every site in a distributed environment and is responsible for receiving intelligent tokens, interpreting their CNs, processing transitions of CNs, updating information in tokens, and forwarding them to another agent.

1.2 Scope of the Report

A version of a job that will contain local data, control information, constraints along with a record of the job's history can be created which will permit the distributed execution of the job to automatically be carried out. A CN and its marking supply the control information as well as constraints on how the job's data may be manipulated. The major purpose of this report is to describe the concept of a CN and also the implementation of a prototype system which shows that execution of jobs can be controlled through the use of a CN.

Several types of constraints, including the timing restrictions which must be observed by a job, are implemented as predicates in this prototype system. A job that is sent to a CN agent for service should be processed in a reasonable amount of time. Whenever this time limit is exeeceded, a CN agent may try to route this job to some other alternative procedure.

For an office system to be computerized, jobs have to be electronically represented in addition to automating the routing of jobs. The electonic versions of jobs are designed as intelligent tokens, which have control information, to enable them to be delivered automatically to the next site on the routing list in accordance with its Control net. All the operations and data which are needed for a job will be stored either in the CN or inside other structures in the intelligent token. As a potential application, this system can be applied to office information systems, which are inherently distributed systems, to help automate the distributed processing of jobs.

A prototype system which contains CN agents which manage the routing and synchronization of jobs has been implemented on the 3B computer network in the Computing and Information Science Department at Kansas State University. The 3B Computer Network interconnects certain UNIX[1] system-based host computers and other network compatible peripheral devices to form a local computer network which use the TCP/IP protocol. The TCP/IP computer network protocol serves as the basis for our CN agents' intercommunication. The synchronization information for a job identifies the sequences of procedures that are allowed to operate on a job. Routing and synchronization requirements for a job are specified via a CN.

1. Trademark of Bell Laboratories

The original concept of our proposed model comes from papers by Dr. McBride [McBR83], [McBR87]. An electronic version of a job as well as an implementation of a generic CN agent has been developed in our system. A job's designer who uses this tool only has to concentrate on designing the specification of the job at a high level of abstraction. The CN agent does the actual routing and synchronization of jobs in the system automatically.

1.3 Report Organization

This report is organized into four chapters. Chapter 1 gives an overview and justification of this report. A survey of related articles from the literature is presented in chapter 2. Details of the functional specification for our proposed system are presented in chapter 3. In chapter 4, we discuss the value of this work and make suggestions for possible extensions. Finally, two appendices at the end of this report provide supporting documentation in the form of a flow chart and the source code for the system which was constructed.

CHAPTER 2
REVIEW OF THE LITERATURE

2.1 Computer Network and Distributed System

A computer network interconnects separate computers. The purpose of a computer network is to allow users at one machine to utilize resources on other machines. A user may log onto a machine at some (possibly distant) site and transfer files from one machine to another. Furhermore, a single user's problem can be divided into subproblems so that these subproblems can solved on different machines.

In this report, we describe a model for specifying and controlling the routing as well as the synchronization of job processing in a distributed environment. A distributed system can be considered a special case of a network, one with a high degree of cohesiveness and transparency [AND81]. A user-friendly distributed system has a system-wide operating system in which services are requested in a uniform way. The distributed system should be location transparent; users of a distributed system should not have to be aware that there are multiple computer systems being used. Resource utilization such as the allocation of files to disks as well as movement of files between where they are stored and where they are needed should also be hidden from the user. Consequently, most of the functions in the system must be performed automatically without the user's attention. In general, a distributed system should behave like a centralized system to its users. For the sake of simplicity, we will not distinguish between computer networks and distributed systems in this report.

The benefits of a computer network can be listed as follows:

1) allows software programs, data, and hardware resources to be made available to anyone on the network,

2) has alternative resources available to increase reliability in the network; when a single computer in the network fails, users can often be accommodated elsewhere,

3) provides a powerful communication medium between computers, especially for local computer networks,

4) increases system performance; some (or all) of the processors can be dedicated to perform specialized functions. Much of the software complexity that is associated with large mainframes can therefore be eliminated.

The prototype system has been designed to accomodate all these advantages.

2.1.1 Distributed System Prototype

The primary intent of this work is to find a good model which will handle the problems of concurrency and distributed control for jobs executing in a distributed environment. A Petri net-based model which matches our requirements for a good model can be found in [McBR83] and [McBR87]. We have attempted to design a prototype distributed system which uses this model as its basis. The Petri net model given in [McBR83] and [McBR87] is extended to model the equivalent of a control program. This extended Petri net, called a Control net (CN), is used as a control mechanism that utilizes intelligent token(s). An intelligent token contains control and data information that is necessary for executing a job. Furthermore, a process which is referred to as a Control net agent is present in each machine node. Its primary responsibility is to execute jobs that are

delivered to the site; the agent accomplishes this task by selecting a transition to execute in accordance with the current marking of the received CN. This extended Petri net satisfies our requirement of modeling concurrent processing and distributed control. A prototype based upon this model has been implemented to run on a small computer network; namely, the 3B computer network at Kansas State University.

We will now proceed to consider the basic elements of Petri nets before looking at their extensions.

2.2 Definition of Petri Net [PETE77]

Petri nets are useful for modeling the states of a system. Modeling can be accomplished by marking the nodes of a graph with tokens. A Petri net can be described by the basic properties of its elements, markings, and firing rules.

2.2.1 Elements of a Petri Net

A Petri net is a directed graph[1] with two types of nodes: places and transitions. Places are represented by circles and transitions are represented by bars. The places of a Petri net can be used to represent conditions, while transitions may represent actions or events. Place nodes and transition nodes are connected by directed arcs, and the two different node types must alternate on any path in the graph. A place that leads to a transition by an arc is called an input place. A place that is connected to a transition by an

1. Petri Nets are directed multigraphs, but we will only deal with directed graph in this report.

arc leading from the transition to the place is called an output place for that transition. Places in the net may be occupied by markers called tokens that are shown as dots. The presence of a token in a place means that a certain condition holds at that place. Places, transitions, tokens, and arcs form the basic elements of a Petri net. A simple Petri net is shown in Figure 2.1 in which all of the places and transitions are labeled.

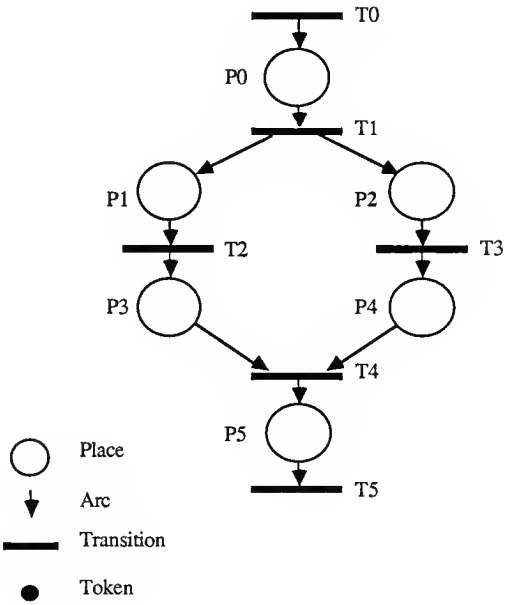


FIGURE 2.1

A Simple Petri Net

2.2.2 Marking of a Petri Net

A marking is the collection of tokens which are found in the places of a net. The number and position of tokens in a Petri net may change during the net's execution. Markings of a net can be used to indicate which transition(s) are enabled at a certain time. Hence, the state of a Petri net is given by its marking. An example of a marked Petri net is shown in Figure 2.2.

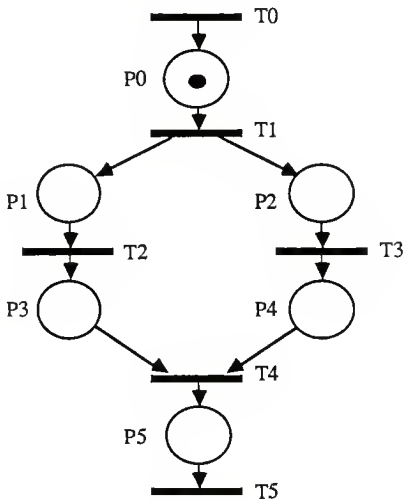


FIGURE 2.2

A Marked Petri Net

2.2.3 Firing Rules of a Petri Net

The number of tokens as well as their distribution in a Petri net controls the execution of the net. A Petri net executes by firing transitions. A transition may fire if it is enabled, and the firing of transitions changes the arrangement of tokens in a Petri net. When each input place to a transition contain at least one token, the transition is said to be enabled and may therefore fire. Whenever a transition fires, a token is removed from each input place of that transition and a token is added to each output place of the same transition. All these firing actions occur as a single indivisible operation. Transition T0 is a source transition because it has only output place without any input place. T0 can fire at any time; the result of one such firing is illustrated in Figure 2.2. The result of firing transition T1 of Figure 2.2 can be seen in Figure 2.3. Figure 2.4 shows the new marking that is produced after firing both T2 and T3 of Figure 2.3. Firing transition T4 of Figure 2.4 yields the marking of Figure 2.5. Transition T5 becomes enabled when a token in place P5 and it can fire. Since transition T5 has no output place, this transiton is called a sink transition. Whenever T5 fires, it merely removes a token from it's input place. An enabled transition may fire at any time. Therefore, if more than one transition in a net is enabled, the next transition to fire is chosen at random (see Figure 2.2 and 2.3).

This feature of randomness in Petri nets reflects the fact that in real life situations several things can happen concurrently. The apparent order of the occurrence of events is not unique, but rather reflects a possible observation sequence for the concurrent events. Even though Petri nets are very simple in concept, they have powerful representational properties which use nondeterminism to show concurrency in a simple and natural way.

Thus, Petri nets are ideal for modeling systems with distributed control in which multiple processes can execute concurrently.

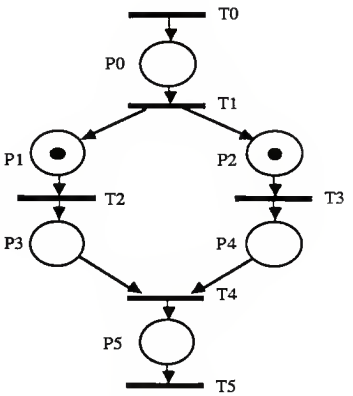


FIGURE 2.3

A Petri Net After Firing Transition T1

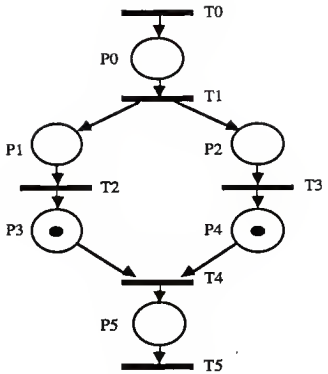


FIGURE 2.4

A Petri Net After Firing Transitions T2 and T3

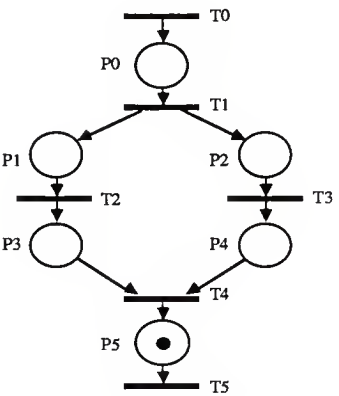


FIGURE 2.5

A Petri Net After Firing Transition T4

2.2.4 Time Limitation of a Petri Net

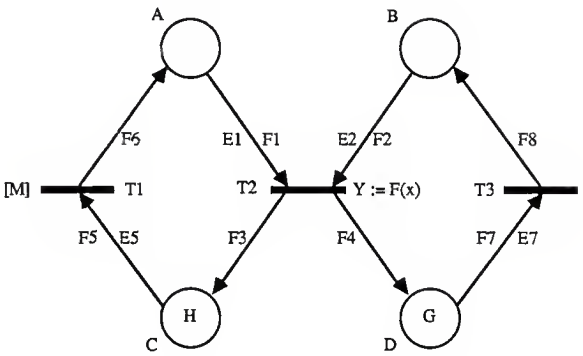
The normal concept of time is not involved in Petri nets. However, researchers were concerned about the problem of overcoming this limitation. As a result, Time Petri nets (TPNs) [MER74] were created to allow for the representation of timing knowledge in a Petri net-like model. The TPN is defined by a Petri net where each transition has two different specified times. The first denotes the minimal time that must elapse from the moment all the input conditions of a transition are enabled until this transition can fire. The second time denotes the maximum duration of time that the input conditions can remain enabled before the transition must fire. These two times can be used to provide a measure the range of execution times for a transition. The TPN model contains the Petri net as a special case in which all transitions have a minimal time of zero and a maximal time of infinity. If the timing of transitions or order in which transitions fire is important, a TPN can be used to model the system under consideration.

2.2.5 Numerical Petri Nets

Numerical Petri nets (NPNs) are a generalization rather than an extension of Petri nets. NPNs [SYMO80] incorporate some changes into the concept of a Petri net. For example, NPNs can have several types of tokens, and each token type can have several attributes. Places in the net can hold any number of different types of tokens at the same time. Individual arcs of a net have independent enabling and firing conditions. A memory reference enabling condition can be specified as a predicate condition on the memory variables and every transition may have its own associated predicate. Also, two firing rules are defined for each transition. One rule defines for each input arc what token(s) are

removed from the respective input place(s) when a transition fires. The other rule defines for every output arc what token(s) are put into the respective output place(s) when the transition fires. If in addition to the memory reference enabling condition of a transition, all the enabling conditions of all the input arcs of a transition are true, the transition becomes enabled and may fire. Transition firing actions include the withdrawal of tokens from input places, the addition of tokens into output places, and operations on memory data. NPNs overcome some of the limitations of PNs, such as the existence of only one type of token, so that more diverse practical problems that are based on these extensions can be covered. For example, NPNs can represent systems where the flow of different types of messages is important. As a result, Numerical Petri nets are especially suitable for representing systems, such as communication protocols, where the flow of different types of messages is important.

An example of an NPN is shown in Figure 2.6.



G, H	TOKENS
E	ENABLING CONDITION
F	FIRING RULE
Y := F(X)	TRANSITION OPERATION
[M]	MEMORY REFERENCE ENABLING CONDIITON

FIGURE 2.6

Example of NPN

Source: [SYMO80]

For instance, the enabling condition E5 in Figure 2.6 refers to 'at least one token H residing in the input place C' of transition T1. A predicate condition on memory variable M (e.g. M = 0) is specified for transition T1 as a memory reference enabling condition [M].

When the enabling condition E5 and the memory reference enabling condition [M] are true, the transition T1 is enabled and may fire. Transition T1 has a firing rule F5 associated with the input arc 'defining a token H is to be removed from the input place C' when T1 fires. Also, firing rule F6 is associated with the output arc of transition T1 that 'defines a token H is to be placed into the output place A'.

Petri nets can be considered a restricted class of NPNs in which no memory is associated with the net. Tokens do not contain values in Petri nets. Also, when a transition fires, only one token is removed from each input place and one token is put into each output place. On the other hand, a NPN allows different types of tokens to be present. In addition, NPNs add the enabling condition, specific firing rules, and memory reference enabling condition to the concept of a transition firing in the original Petri net graph. These additions serve to increase human understanding of complex systems and make NPNs more program-like.

2.2.6 Data Flow

The Data flow model [FILM84] makes use of directed graphs to illustrate the flow of data and control which must occur between instructions in a system. The arrival of operands are used to trigger the execution of an operation or instruction in the Data flow; thus this model is data driven. Also, data representing the result of one operation can be passed as an operand to another instruction.

In contrast with Petri nets, "places" in this model do the work (e.g. assignment) while "arcs" (edges) serve as storage. Places representing instructions are activated by data

tokens. Data flow graphs distinguish between data-carrying paths and control paths. Data paths carry the data values resulting from a computation. Every Data flow token has some value in a specific data type such as integer, boolean, or character. Control paths, on the other hand, carry control values (booleans) that "open" and "close" values; thereby, regulating the flow of data around the graph. In this way, the flow of control is tied to the flow of data.

A Petri net graph could be represented by a Data flow graph, and vice versa. Both Petri net and Data flow models share the same conceptual basis of modeling change through successive markings of a graph structure. The control flow for a Petri net is hightlighted, while the data computation for a Data flow graph is more prominent. Petri nets are useful for modeling the states of a system and transitions are used to represent major operations which are needed for a system. However, tokens that are passed around a Petri net do not contain any data value information. Even though tokens in the model of [McBr83] do carry control and data information, the information that these tokens convey cannot be seen on the graph. In this sense, a computation modeled with a Petri net is still not as natural as with the Data flow model. The Data flow model broadens the limited capabilities of Petri nets into a mechanism that can perform computable functions. Data tokens in a Data flow graph may have a constant, a parameter name, or a boolean value written next to these tokens making the flow of data more prominent. The Data flow model has been a source of ideas for both computer hardware and programming languages. This model is suitable for the design of both low level computer hardware and programming languages. On the other hand, a Petri net is more suitable for designing and simulating the high level flow of control for a system.

2.3 Survey of Related Work

In the following sections, we will review several articles that are directly related to job execution using Petri nets in a distributed environment.

2.3.1 Formal Verification of Parallel Programs [KELL76]

In this paper, R. M. Keller developed a formal model to represent parallel programs. In his model, place nodes represent points at which an instruction pointer of a processor may dwell and transition nodes denote events which correspond to the execution of particular instructions. Keller modeled the action of transition nodes in a Petri net upon program variables. With his modification, each transition has an associated unary predicate and a function that is defined upon the program variables.

According to Keller, the state of execution of a parallel program consists of both a control state which represents the vector of place variables (marking) and a set of data states which reflects the vector of program variables (local data). The control state of a program at any point in its execution is given by the number and location of all tokens residing in the Petri net. Each token in the net corresponds to an instruction pointer. The data state of a program is equivalent to the current values of all its program variables.

The existence of a predicate indicates a precondition that is associated with a transition which must be true before that transition can fire. The firing requirements of a transition is defined as follows:

1) Each input place of the transition must have at least one token present,

2) The predicate associated with a transition must be true.

Firing a transition in Keller's net causes the following events to occur:

1) a token is removed from each input place,
2) a transformation upon the program variables is performed in accordance with the function specified by the transition; this operation has been added to the usual Petri net firing operations,
3) a token is added to each of the transition's output places.

By allowing arbitrarily many instruction pointers (or processes) to execute the program, this model has the capability to represent an infinite set of control states.

2.3.2 Modeling Jobs in a Distributed System [McBR83]

In this paper, McBride and Unger described a method for modeling the execution of jobs in a distributed system. A model was presented to depict the control and information flow of a job in a distributed processing environment. Individual Petri nets define the procedures that are available in the system, and a "Control" Petri net is used to define how a job is to be executed by the system. The Control Petri net oversees the execution of procedures that are available in the distributed system. The state of a job is defined by the location of token(s) in the Control net. This can be thought of as the position of the instruction pointer(s) in a job control program as was done in Keller's model.

The Control Petri net is augmented by attaching system resources (such as files) to the transitions in the net. An arc can be directed from a file where input data is needed by a

transition which utilizes the data. Similarly, when a transition outputs data to a file, a directed arc will go from that transition to the file. In this way, transitions that depend on the availability of data in files are brought into prominent view. The collection of tokens over a net represents the execution state of a job. This information, called the marking of a net, is stored inside an intelligent token. The Control net defines possible sequences of job steps; each job step may be implemented at a different site. The intelligent token migrates through the distributed system to the site where its Control net has determined the next job step can be performed. Therefore, tokens can be visualized as intelligent data objects that carry meaningful information through the system. The Control net is stored as a data structure contained within a token while the token is flowing between nodes, whereas individual Petri nets correspond to the procedures that are requested by the job. These individual Petri nets are triggered by the arrival of intelligent tokens.

Each intelligent token can be thought of as an object that consists of the following information:

1) local data,
2) Control Petri net,
3) marking,
4) a list of capabilities,
5) a history list.

Five major components which are necessary to model the processing of a job in a distributed environment were identified:

1) a structural model for each procedure or function,
2) a structural model of the control program,

3) the status of a job (control and data),

4) global information,

5) data files.

Since the intelligent token contains control information, a site in the distributed system can decide what procedure to perform for a job according to the control information and also which site it should be forwarded to next.

2.3.3 The Representation and Distribution of Knowledge by a Petri Net [McBR87]

In this paper, McBride and Unger refined some of the ideas from their previous work [McBR83]. For example, refinements were made to the Control Petri net and the intelligent token. The new feature in this paper involves the addition of a Control net agent.

2.3.3.1 Control Petri Net

A Petri net can be used to provide a top-down, hierarchical representation of a system. Any action performed by a transition in a net at a high level can be represented by another Petri net. Each net provides the coordination and communication among its actions or underlying nets. The uppermost net in a hierachy of Petri nets is referred to as a Control Net (CN). It enumerates possible sequences of significant events which may occur in a system. Thus the CN can serve as a source for both the routing and synchronization information in a distributed environment.

A set of program variables is used to construct unary predicates. Preconditions and post conditions are constructed by unary predicates which are attached to transitions. A unary predicate is used as a precondition predicate to govern the firing of a transition. Also, unary predicates are used as postconditions after a transition fires. The use of a postcondition predicate results in a test that ensures each job step will meet its specification, consequently increasing the system's reliability.

2.3.3.2 Intelligent Token

Tokens mark the transaction's progress through the system and contain job-related information. Accordingly, a token contains both control information, such as the logical routing, and data information. The key point here is that the intelligent token must contain enough information so that each site can figure out how to react to tokens that are delivered to the site in a distributed system.

2.3.3.3 Control Net Agent

A CN can be attached directly to a token instance and accompanies the token on its journey through the system. Furthermore, the CN acts as a guide for a particular job. A group of cooperating entities, named Control Net agents (CN agents), exists in every site of a distributed system. A CN agent at each site in the system interprets the Control net to oversee processing as tokens pass through the system. CN Agents actually carry out the physical routing, perform the execution of the Control net, and update data information of the token. Details of the routing algorithm and re-routing of a transaction to an alternate module or procedure are transparent to the transaction's CN.

As suggested by its title, this model is suitable for the communication system in a distributed environment . A token has to carry the control message with it so that the next site which receives the token can figure out how to react upon the token's arrival. The advantage of this model is that a job can access all the functions which are supported in the network. This system can execute a job in accordance with its control information as well as the precondition and postcondition predicates of the transitions. The loss of messages on their journey through the system has not been dealt with in this report.

2.4 Summary

In general, all of the referenced articles retain the basic principles, symbols, and modes of operations of original Petri nets given by Petri. The authors have suggested ways to extend Petri nets in order to cover a wider range of situations. Some of these ideas are similar or overlap with each other.

If all of the features (e.g., timing knowledge, types of tokens, input and output firing rules, precondition, postcondition, program variables, intelligent tokens, Control net and Control net agent) could be combined, a model would arise which would become a very powerful tool. However, the resulting Petri net model would become complicated and cumbersome to use. The designer has to decide which features will fit his needs for a particular practical situation.

Control Petri nets which incorporate the idea of program variables and predicates are used in our prototype system. The concept of intelligent tokens is from [McBR83] with many of the other extensions being taken from [McBR87]. The design of this system

consists of three major parts: CN, intelligent token, and CN agent. We proceed to describe the design of our prototype system in the next chapter.

CHAPTER 3
DESIGN SPECIFICATION

3.1 Design Objectives

The purpose of this report is to describe a system that permits the execution of jobs in a distributed environment. A prototype of this system has actually been implemented using a TCP/IP[1] WIN[2]/3B system on a network of AT&T 3B series of computers. Details for the design of this system are discussed in this chapter. Also, a sample of the code for a CN agent can be found in appendix B.

Our prototype system is designed to solve simple numerical calculation (referred to as CALC) problems. An ideal distributed system should not only be able to compute solutions for numerical calculation problems, but should also permit the distribution of messages between different machines so that nonnumerical computation could be performed. For ease of explanation, CALC will be discussed in this chapter instead of other more general jobs or problems. Readers should keep in mind that this simple prototype system can be expanded to solve numerical computations that are more complex than the simple examples given in this chapter.

In general, the goals of our prototype system can be listed as follows:

1) reduce the demand for user involvement after a job has been created,

---

1. Transmission Control Protocol/Internet Protocol is a set of computer networking protocols which allows two or more hosts to communicate.
2. A trademark of the Wollongong Group, Inc.

2) effectively utilize needed resources (e.g., functions and utilities) in a distributed environment that are distributed over multiple locations,
3) raise the reliability of the environment which is provided to users,
4) take advantage of concurrent processing that is possible.

Accordingly, our prototype system has been implemented to meet these goals. After a user submits a CALC job to the system, he is relieved of the responsibility for the job and only has to wait for the result. The user does not have to worry about the location (site) of the functions which are needed by that job. This system will automatically take charge of routing the actions for each job. A simple error recovery strategy is used to ensure that tokens are not lost.

Concurrent processing is approached by dividing a job into multiple subjobs or tasks. If these tasks are logically independent from each other, they can be executed simultaneously in an effort to cut down on the total run time of a job; thereby, making the execution of a job more efficient.

Three main steps are involved when designing this system to perform a task, namely, the design of: the CN, the intelligent token, and the CN agent . The actions of the CN agent depend on how the CN and intelligent token are designed. After the CN and intelligent token are designed, the actions of the agent can then be coded.

The set of actions locally available to a CN agent dictates the tasks that can be performed at that agent's site. If the condition predicated upon an action's outcome cannot

be met or no desired function is available, the task to perform will be transferred to another site. The coordination of actions for a job at a site are taken care of by that site's CN agent.

3.2 Design Specification

We designed of our system based on the model which was described in [McBR83] and [McBR87] with some slight modifications. In the remainder of this chapter we provide a description of both the design and implementation of this system. This description logically falls into three parts. First, we describe the CN representation of a job. Next, we look at the components that comprise an intelligent token. Lastly, we discuss the flow of control which a CN agent follows, i.e., the actions that a CN agent performs on a task and the order in which they are done. Finally, some problems concerning error recovery are mentioned, and a simple error recovery strategy which is used in this prototype is also described.

3.3 The Control Net

Both internal and external representations of a CN are needed in this system. The external CN is designed for use by humans, while the internal CN is designed to be used by machines. An external CN is generated by the user to describe the processing requirements of a particular job, while the internal CN provides the representational structure used by computers. The internal CN contains only the structure of the external CN and so captures static properties of the external CN. On the other hand, the sequence of markings recorded by an external CN capture the dynamic properties of the net; these dynamic properties are not recorded in the internal CN. The marking is recorded separately

from the internal CN as a distinct component of an intelligent token. This internal representation is stored in such a way that a CN agent can recognize it.

3.3.1 The External Control Net

3.3.1.1 Use of the External Control Net

Users of this prototype system must do two things before the system takes over the processing of a CALC job. First, an external Control net which represents the control flow of a particular job has to be manually created. The external Control net (CN) is an extension of a Petri net, which permits predicate conditions to be associated with transitions. Also, the maximum period of time which may elapse before a transition times out can be specified for each transition. Secondly, an external CN must be manually translated into an internal CN. After the internal CN of a job is created, that job will continue to be executed by the system until the job is either done or it fails. In the prototype system, users need to manually transform an external CN to an internal CN for each job. In order to make this prototype system more complete, a graphical editor should be built. This graphical editor allows the creation of an external CN which is later automatically transformed into an internal CN. We will first look at the design of an external CN.

A user must first design an external CN according to a job's specification. Some basic knowledge of Petri nets is assumed in order to effectively design the flow of control for a job. We assume that the user will design a correct and efficient external CN. Hence, our prototype system does not check for the correctness or efficiency of the external CN. By correctness, we mean a CN does not have any simple graphic mistakes nor any illegal

control flow for a job. This system can reduce the total run time by performing tasks in parallel, thus, making the execution more efficient. By permitting the 'fork' transition, concurrent activities among tasks can be accomplished. Users should therefore take advantage of the fact that this system can handle the concurrent execution of a set of tasks. It is the user's responsibility to design an external CN with (a) fork transition(s) whenever it is effective to execute tasks concurrently.

An intelligent token is used to represent a task. A token at a fork transition will split into several sibling tokens or 'tasks'. These sibling tokens can then be executed concurrently, because they are independent of each other. A JOIN, also called MERGE transition, is used to combine several sibling tasks or intelligent tokens into a single task.

3.3.1.2 Sample External CNs

Next, we will show two different examples of external CNs.

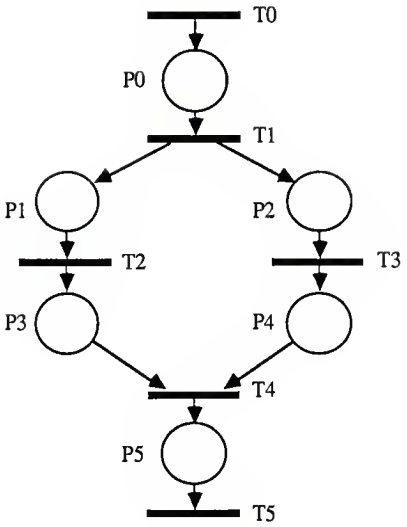
Example 1. (a + b) * (c - d)

Two subexpressions, (a + b) and (c - d), are considered to be independent of each other whenever these expressions have no data in common. Thus, these expressions can be evaluated simultaneously. We can design the external CN for this type of example as given in Figure 3.1a. In the initial state, the source transition T0 will deposit a token into place P0. Since no more than one token is allowed at any place in a CN in our prototype system, the source transition can produce only one token at a time when its output places are empty. In the state shown in Figure 3.1b, transition T1 is enabled and can therefore fire. After transition T1 fires, two different markings are created to represent the two sibling tokens or tasks that are deposited into places P1 and P2. At this time, the two tasks become

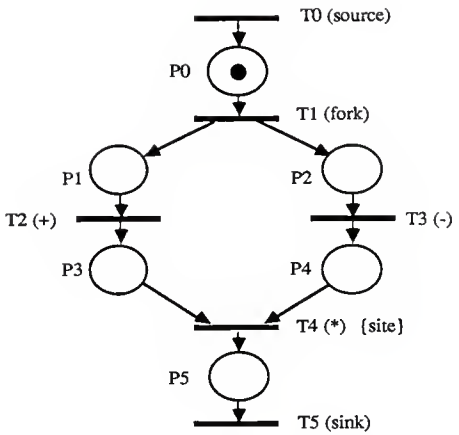
independent of each other. Figure 3.1c and 3.1d show the two tasks that are initiated. Only one copy of this type of CN is stored on each site even though multiple instances may exist in the system.

Tuple (1,0,0,0,0,0) represents the initial marking for the net of Figure 3.1b. Each item in a marking represents the number of tokens at a place in the CN; the *i* th item in the marking corresponds to the number of tokens at place Pi. A '0' in place Pi means that there is no token in that place, while a '1' means that there is a token present at that place. In our prototype system, the maximum number of tokens at any place is limited to 1. The markings for the two sibling tasks right after transition T1 fired are (0,1,0,0,0,0) and (0,0,1,0,0,0). Since these two sibling tasks are independent of each other at this moment they can execute concurrently.

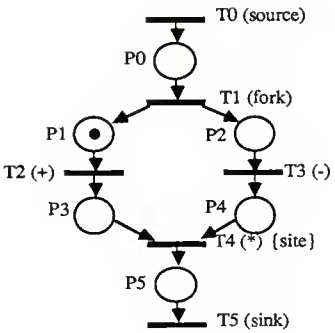
Before the 'multiplication' operation at transition T4 of Figure 3.1b can be performed, tokens corresponding to the results of the 'plus' and 'minus' sub-expressions must be made available on the same machine. One way of accomplishing this goal is to let the user specify a particular machine that is going to execute the '*' operation. A transition can be qualified to specify which site is to perform an operation by enclosing the selected site inside a pair of curly brackets.The curly brackets can be omitted if no specific site is chosen by the net's creator . In our example, a site has been specified by the user at which transition T4 should be executed in Figure 3.1b. Consequently, the CN Agents will forward the results of the '+' and '-' operations to that particular site.



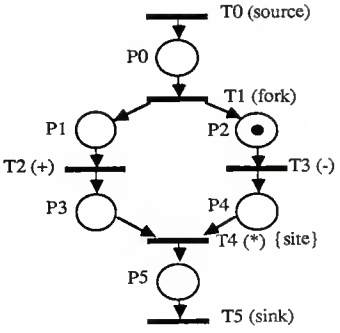
(3.1a)



(3.1b)



(3.1c)



(3.1d)

FIGURE 3.1

A Sequence of Marked CNs for Example 1

A system can be designed to let users enter a list of sites that are capable of performing a 'MERGE' transition (for example, transition T4 in Figure 3.1b might have {site} replaced by {site 1, site 2, ..., site n}). A list of sites at which an operation can be performed is used to increase the reliability of this system. In the event that the first machine in the list goes down, those agents which execute the '+' and '-' operations can send their results to a second site in the list, and so on. This feature has been left out in our prototype system.

Preconditions and postconditions can be added to the net of Figure 3.1b in order to illustrate the additional power of CNs. For instance, a user might only want the action of the '*' operation to occur after a particular time, say 3:00 pm. In this case, a precondition can be specified for that transition with a predicate [>=15:00] written right above the transition as in Figure 3.2. This predicate permits transition T4 to only execute on or after 15:00 for the current day. Users who want an integer result from transition T3 can use a postcondition predicate which has the form [result in integer], written below the transition, to restrict the result of this operation to be an integer (See Figure 3.2 for an example). If the result from T3 is not an integer, the agent which was currently responsible for performing T3 has to re-route the task until the result of the requested operation (-) becomes an integer. Re-routing a task involves choosing another version of the desired function, and may involve sending the task to another agent. This entire process is transparent to the user.

The Petri net of Figure 3.2 demonstrates the additional types of contraintswhich can be made to Figure 3.1b by the user.

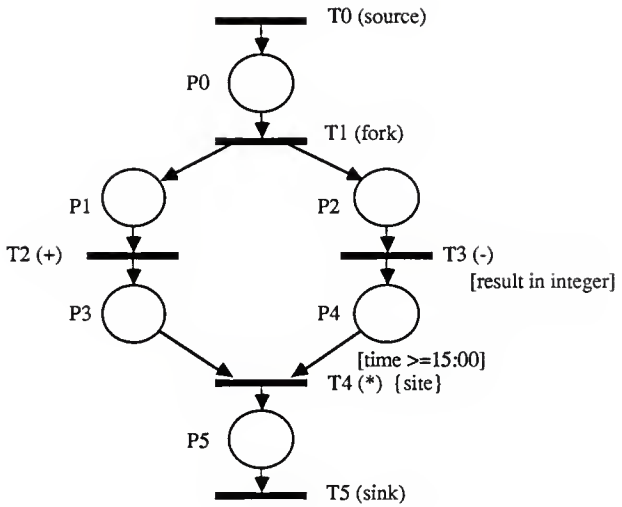


FIGURE 3.2

A CN With Predicates

The Petri net graph of Figure 3.2 can be used as an isolated CN that represents all of the work a user has requested or, alternately, it may represent a subgraph of some other CN. Once a CN gets more complicated, the details of any action which is performed by a transition can be represented by a subnet. In this way, a hierarchy of CNs can be provided; each action in a CN may require a sequence of operations which is carried out by another CN. If the net of Figure 3.2 is a first level CN, the source transition will represent a job entering the system while the sink transition stands for a job terminating. The destination site(s) for a job will be in accordance with the needs of the user. A job may have just a

single destination site or multiple destination sites. In order for the graph of Figure 3.2 to be a subnet corresponding to the effects of a transition in the first level CN, the source transition will represent the invocation of that transition, while the sink transition stands for the completion of that transition. For this graph to be a subnet, the sink and source transitions must be at the same site. In the prototype system which has been developed, the actions which are named in a CN are not implemented as other CNs.

In order to simulate a database environment, the data is stored in different files in our prototype system. Data files can be kept on either the same machine or different machines. The data which are needed for a transition will be requested only when the transition is enabled. In this way, the value of the data will be the most current updated value.

Example 2. (a + b) * (b - d) / (e - f)

The sub-expressions (a + b), (b - d), and (e - f) can be executed concurrently because they do not modify any common data. However, the 'multiply' and 'divide' operations have to be executed in a left to right order for this particular example to produce the intended results. Furthermore, both the '*' and '/' operators require results that are derived from different sub-expressions before proceeding. One way to accomplish a merge transition is to have the users of our system explicitly specify sites that are to perform '*' and '/' operations. The external CN for this type of example can be designed as shown in Figure 3.3:

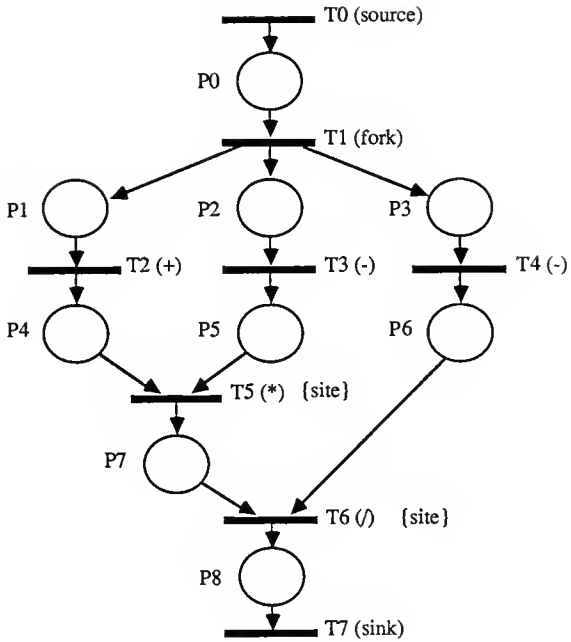


FIGURE 3.3

A Sample CN for Example 2

Thus far, we have shown some simple calculation operations. Other desired actions can be much more complicated than the calculations we have shown, and may involve more complex procedures or modules. If the desired procedure or module required by an action does not exist in any site of a distributed system, the user has to develop that particular

module or procedure and store it at a site before a job requiring that function can be executed.

3.3.2 The Internal Control Net

To make sure that agents at every site understand a CN, an internal form of a CN (internal CN) is employed; this type of CN can be derived directly from the external Petri net graph which was described in the previous section. This internal version of a CN must be created in such a way that it can be recognized and understood by a computer rather than users. The internal CN used in this prototype is produced by a simple transformation on a corresponding external CN. The internal CN is a data structure which records the static properties of the external CN. These static properties include the relationships between the places and transitions of a net as well as predicates associated with the transitions of a net. For example, information such as the set of output places associated with each transition and the collection of input places for every transition depict the relationships between places and transitions.

We now discuss details of the internal CN that was designed to represent the CN shown in example of Figure 3.1a. An internal CN's data structure depicts the relationship between places and transitions in a net and depicts the predicates on transitions. Since an internal CN contains only the static properties of an external CN, the structure of the internal CN will not be modified during a job's execution in the system

The CALC problems used in our examples are assumed to be requested quite often by users. For efficiency, the prototype system preserves a copy of the internal CN for each

type of job at every site instead of passing the static structure of a job through the communication links. In this case, only dynamic information (e.g. the marking, local data, and routing history) that will be modified as a job's execution is passed between sites. Thus, for a frequently used job, its token will not carry its internal CN. It makes sense to store the static structure of a CN on every site so as to save the cost of transferring data. On the other hand, a job which is executed only rarely does not need to store its static structure at every site. Thus, in the case of infrequently executed jobs, the expense of initializing and maintaining the internal Control net at each site is spared. Instead, the internal CN is communicated as necessary between sites as information in the intelligent token. Users have to decide whether to store the static structure on every site or to pass the static structure as part of the token between sites.

An intelligent token in our prototype system is an object that contains: the marking of an internal CN, any local data, and a routing history of a job. Thus, a token is reduced to the minimal amount of information that is required for execution. This reduction of information decreases the cost of transferring messages between sites and the probability that transmission errors will occur during data transfer is also decreased.

Certain control information, which is not related to the CN also has to be included inside a token. This information includes the token's id and an index for the next position of the routing history. Further details of the intelligent token used in this prototype sytem are described in the next section.

3.4 The Intelligent Token

3.4.1 Types of Files

In this report, we regard the token in a Petri net as an intelligent communication object that contains both control information and local data. Control information for a token is provided by the marking of a CN which the token contains. The local data of a token holds values required by the corresponding job. A token is stored as a file in our prototype system. Files are used as the unit of communication in this prototype system and so agents communicate with each other by transfering files. A token may be treated as either a MSG (message) token or a DATA (data) token. The need for both MSG and DATA tokens arises because of MERGE transitions which require multiple tokens to be input, but which ouput at most one token. Both MSG and DATA tokens contain the marking of a CN, local data, and routing history. A complete listing of the components of a MSG token is given in section 3.4.2.

MERGE transitions are handled by our prototype in the following manner. Only the token to arrive at the first input place for a MERGE transition will be kept as a MSG token (this token will become the master token). All of the master token's siblings which arrive at the MERGE's other input places become DATA tokens. The agent will use the master token and its control information when the MERGE transition is fired. The agent will perform this MERGE transition by collecting information from all of the data tokens in accordance with the MERGE transition and inserting this information into the MSG token which is output.

In addition to token messages, the prototype system utilizes two other types of messages, namely, the request (REQ) and acknowledgement (ACK) messages. A REQ message is used by an agent to request a particular variable's current value from the agent at the site where the variable is located. An ACK message is used to inform another agent that a token (including the MSG token and DATA token) has been successfully received or that the requested operation has failed.

3.4.2 Components of Intelligent token

Before discussing the actions of a CN agent, we will first explain the components of an intelligent token. This type of token is an object which is used as an information unit for communication between sites. Basically, intelligent tokens carry the marking of a CN, local data, and a routing history. Moreover, a token in this implementation contains six different items. The items are the **id of a token**, **token's type** (either MSG or DATA), **next location in history**, **marking of a CN**, **local data**, and **routing history**. Each item in a token can be described in detail as follows:

1) The first item carried by a token is the id of that token. This id consists of a four digit integer in which the first digit (i.e. leftmost digit) stands for the type of CN. Several types of CNs may exist in a distributed system. Ten types of CALC CNs ranging from type 0 to type 9 in decimal, are allowed in this prototype system because only one digit is used to represent each type. The second digit stands for the instance number of a type of CN. Copies of a token for the same type of CN are considered instances for that CN. Each type of CN ideally can have an arbitrarily large number of instances. However, this particular prototype system can only handle up to ten instances for each type of

CN; since only one digit is used. Extending the number range for CN types and instances according to any particular need is easy to accomplish by simply adding more digits. All instance's ids from the same type of CN have an identical first digit but a different second digit.

The third and fourth digits are taken together to stand for the current place which is marked in a CN. These two digits are appropriately named 'place number' for ease of reference. The place number is mainly designed for programming the FORK and MERGE transitions. A token before a fork transition will produce multiple copies of sibling message tokens. Each message token is distinguishable from its siblings by the place number of its token id. This place number is later used to distinguish between multiple copies that need to be merged. The token ids of these copies will have the same first and second digits, but different third and fourth digits. The first and second digits of a token does not change, but the third and fourth digits are dynamic and therefore depend on the current place that the token resides in. Thus, the last two digits of the id for a token changes each time that a transition successfully fires. Each sibling token will have a different marking, local data value, and a routing history after a fork transition. Since two digits are sufficient to show the maximum number of places in our examples, only two digits are used to represent the place number in our prototype system. Therefore, place numbers can range from '00' to '99'.

An example of how a simple token id is used is shown in Figure 3.4. Numbers which reside inside the circles of a Petri net are called token ids. We

can see from the first and second digits that this token is a type 1 CN and each time the token is used, it refers to the first instance of this CN type. The last two digits corresponds to the place where a token would reside.

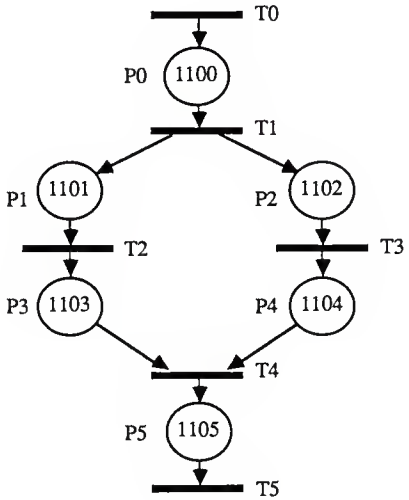


FIGURE 3.4

Example of the Token Ids

2) The second data item in each token is used to specify the token type; it can either be MSG (message) or DATA (data). Even though a token type can be determined from its data file name, information about the type is easily kept

inside a token. Doing so allows the MSG and DATA token to be distinguished through the use of either the file name or this field.

3) The third item is called 'next location in the history'. This is an index to the position in the routing history entries where firing information for the next transition is to be placed.

4) The fourth item keeps a record of the current marking for a CN; this is implemented as an integer array. This array will keep track of existing token(s) in every place of a CN. At this time, the allowable values of the marking are "0" and "1".

5) All local data that is needed for a job must also be stored in the intelligent token that corresponds to a job. Local data are used mainly for storing intermediate results that are produced during a job's execution as well as input data that are needed for an operation.

6) A list of the routing history is the last item in a token. The routing history is used to keep a record of the processing which a token has undergone. The information in the routing history consists of the token's id, transition number, name of operation, site where the operation was executed, and the time at which the operation occurred. Only successfully executed transitions are kept in the history list.

A sample token with all its items is shown in Figure 3.5 (this token corresponds to the CN of Figure 3.3). This job is started with a token having the following information, **2100 m 0 100000000 [empty local data list] [empty history list]** along with an internal CN at every site. When the job is completed, its final token has 2108 as id. Looking at this final id in Figure 3.5, we see that: this job belongs to the second type of

CN; the instance number of this job is '1'; its current place is P8; 'm' is the token type signifying that this is a MSG token; and its 'next location in history' is pointing to position '9'. This marking shows that the job has completed. A list of local data which contains the result and input data follows the marking of the CN. Finally, a routing history list makes up the last item of the token. In the routing history, '**2100 0 source november Wed-Apr-13-12:32:07-CDT-1988**' is the first line in the history. Upon closer inspection of this entry, one can see that 2100 indicates the token's id. The next new location in the history is 0 which means that its own history will be placed at position 0 in the history. The operation which was performed was 'source', on the machine 'november' which fired the source transtion, and finally, 'Wed-Apr-13-12:32:07-CDT-1988' is the time stamp for when this transtion fired.

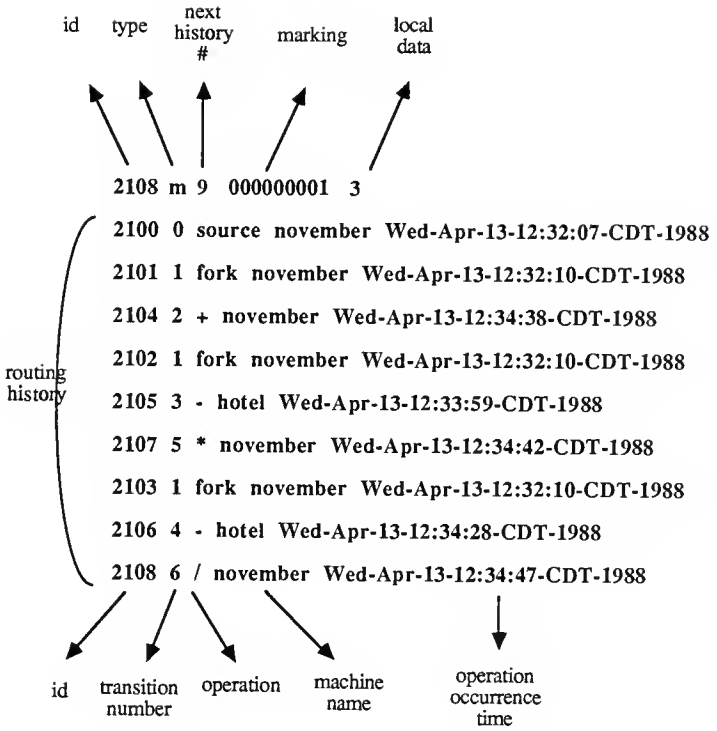


FIGURE 3.5

An Example of a Token

The reader is referred to appendix B for more information concerning the actual data structure of the CN.

3.5 The Control Net Agent

Each machine in a distributed system should have a copy of a controller process which is referred to as a CN agent. This agent is locally in charge of the actual management and execution of jobs in accordance with a job's control specification. The CN agent is responsible for receiving tokens and then performing the next executable transition of a token whenever possible. If the result obtained from firing a transition meets this transition's post-condition, the agent updates the marking, local data and routing history, and ensures that the token is transfered to agent(s) responsible for the next operations to be done.

On the other hand, if the CN agent has tried all the possible alternative solutions to complete an operation at its own site, and has still failed to meet the postcondition associated with a transition, the token will be transmitted to another site that supports the desired type of function. In case no available site supports this function, a failed information containing the message 'FAILED to continue' will be sent to the site where the job originated. Any re-routing of a token which is required for processing is transparent to the user since only transitions that are successful are recorded in the token's routing history. In general, the location where a task is performed is not important as long as the task is successfully completed. However, for some cases the user can name the specific site where the operations are to be performed for a transition. For instance, the user explicitly specified the location where the transition is going to perform a 'multiply' operation in our first example (see Figure 3.1b).

For efficiency purposes, CN agents could be directly incorporated into the operating system (OS). However, CN agents are implemented in the prototype system as user-level modules. This was done so that the integrity of the OS would be preserved even in the presence of a faulty agent and also because it is easy to modify and expand agents that are single modules. We can view the CN agent as a daemon server process.

The processing which a CN agent must undertake is recorded in its Time Table. A Time Table is a data structure which maintains a list of information about tokens that are waiting for an acknowledgement message, a DATA token, or variable values to be retrieved from another machine. This list of information includes: the token's **id**, the **time** at which a token was last sent out or put to the list, the **site** which this token was sent to if any, and a copy of the **local routing history** for the intelligent token which activated the current transition. The CN agent will periodically check whether any tokens in the Time Table have timed-out, or any new in-coming message tokens or requests for a variable from other agents has arrived. A token that has timed-out will be re-processed in preference to a new in-coming token.

3.5.1 Major Actions of a CN Agent

A CN agent is responsible for three major actions: periodically checking the Time Table to catch timed-out tokens, receiving any new incoming tokens, and generating an ACK to inform the other agent of a particular token's receiving status. These three actions will be discussed in detail in the next sections.

3.5.1.1 Checking the Time Table

An agent will periodically check the Time Table in order to catch tokens which have timed-out. A token that has exceed a time limit while waiting for an ACK is resent by the agent to the same site again. An agent can only resend tokens, which have previously timed out, to the same site for a certain number of times. This maximum number of retry attempts is specified beforehand to the system. A site is assumed to be currently down if the number of tries to it reaches this maximum number. Whenever the maximum number of retries is reached for a site, the agent must check if any other site supports this type of desired function according to a capability list for all machines. If another site does support this function, the token will be re-routed to that new site with the current transition still marked as enabled. A token that has timed-out while waiting for a DATA token will again be reprocessed. After the number of retries for a token which is waiting for an DATA token reaches the maximum number, a 'DEAD' ACK will be generated in our prototype system. A token that times out while waiting for a variable's value to be transfered from another site will make the same 'REQ' message again. After sending the maximum number of same 'REQ' message without any success, the token will be re-routed to an alternative site which also holds a value for that desired variable.

3.5.1.2 Receiving Incoming Tokens

Whenever a new MSG token has been received, the agent will try to perform the next executable transition. An agent in our prototype system will create another process, called a child process, to actually execute the operation. The agent will then go to sleep for a while. If the child process has not finished the operation when the agent wakes up, that

agent will re-route the token to some other version of the desired function (module). If the result of this function meets the transition's postcondition, this token will continue to process until either the token completes all of its required processing or this current site can no longer support functions requested by the enabled transitions of the CN. On the other hand, if the result which is obtained fails to meet the postcondition of a transition and no alternative module is available, the agent will consider this token to be dead. A 'FAILED to continue' flag is sent to the original site of the job as well as any site that is waiting for an ACK from the current site.

3.5.1.3 Generating ACKs

The three types of acknowledgements (ACK) uses in this prototype system are 'NOTICE', 'DEAD', and 'DONE'. A 'NOTICE' ACK is used to indicate successful receipt, a 'DEAD' ACK indicates that processing of a token can no longer continue, whereas a 'DONE' ACK indicates that a job has completed. A 'NOTICE' ACK will be generated after an agent has successfully received either a MSG or a DATA token from another site. A token is considered successfully received if the first enabled transition of its CN has fired successfully. For example, an agent at site A who has tried all possible local operations to accomplish a requested function without success will try to transmit this token to another site that supports the desired function. After the CN agent on site A has sent out a token to site B, site A's agent will await a 'NOTICE' ACK from site B's agent. This ACK information must be received within a certain time period. After site A receives an ACK ('NOTICE' type) from site B, the copy of the MSG or DATA token in site A will be purged. Site A's agent will try to send the same token to site B again if no ACK is received within a certain time period. If site A's agent has tried all possible ways of re-routing a

token to alternative agents without any success, a 'DEAD' ACK will be sent to the starting machine. After an agent completes a job, a 'DONE' ACK will be sent to the starting site as well as to the previous site which is waiting for an ACK from the current site.

While performing an executable transition, an agent may need a certain variable's value from another site. In this case, a REQ message is sent to the site that holds the required variable. The agent who owns the requested variable has to send the value of that particular variable to the requesting agent. The above method was used in our prototype system. There are certain tradeoffs when using this approach. It makes sense to send data across the communication line if the requested variables are small in size. However, large amounts of requested data will cost too much to transfer. In the latter case, it might be better to transfer the desired function or module to the site which holds the required data. However, the best solution is to keep both the data and desired function at the same site.

After reviewing the main actions for a CN, we can now present the pseudo code of the control flow for a CN agent as follows:

3.5.2 Pseudo Code of a CN Agent

```
/* pseudo code of the driver and main module */

main driver
{
    initialization;
    FOR NOT system crash
        DO
            IF no incoming data
                THEN
                    sleep 60 seconds
            FI;
```

```
                IF incoming data is an acknowledgement
                    THEN
                        update [Time Table]
                        do some clean up
                FI;
                IF incoming data is a request
                    THEN
                        forward the value of a variable
                FI;
                IF incoming data is a message
                    THEN
                        insert this token id to [Ready List]
                FI;
                check the timed-out table;
                IF there are any messages which have timed-out
                    THEN
                        copy those message's id(s) from [Time Table] to [ready list]
                FI;
                FOR NOT empty [Ready List]
                    DO
                        process the message
                    OD;
        OD;
} /* main driver */

process the message
{
    IF terminate marking
        THEN
            send 'DONE' ack to the start site
        ELSE
            get the current transition number from the marking;
            IF there is a function in this site
                THEN
                    select the module;
                    IF the pre-condition holds
                        THEN
                            do the action
                        ELSE
                            wait until the precondition is true
                    FI;
                    IF the post-condition does not hold
                        THEN
                            re-select the module
                            check the pre- and post- conditions
                        ELSE
                            update the marking, local data, and history of
                            the message
                    FI;
                ELSE
```

```
                    IF the function is supported by some other site
                        THEN
                            send the message to that site
                        ELSE
                            /* no nodes supplied such a function or no nodes
                            supply the desired function which will satisfy
                            the post-condition */
                            send 'DEAD' ack to start site
                    FI;
            FI;
} /* process the message */
```

Notes:

[Time Table] is a list which keeps information about the time of occurrence for each operation.

[Ready List] is a list which keeps the ids of messages that are ready for processing.

A flowchart of the CN agent can be found in appendix A.

3.6 Error Recovery

The CN agent deals with error recovery by sending out an acknowledgement (ACK) message to other site(s) in three situations. A different type of ACK is sent out in each case. The first situation occurs after the first enabled transition of a new incoming token has completed. For instance, the CN agent on machine B will send a 'NOTICE' ACK to the agent that sent the token. The 'NOTICE' ACK is used to inform the CN agent on a particular machine that a certain token has been received as well as the completion of the first enabled transition. This type of ACK is designed to recover from the cases when either a network error occurs or a machine is down.

Figure 3.6 shows a fragment of a CN in which Ti is the first enabled transition of a token. Assume $Ti\text{-}1$ was just completed at machine A and that this token has been sent to machine B. At this point, machine A is waiting for an ACK from machine B. If Ti is successfully fired on machine B then a 'NOTICE' ACK will be sent to machine A.

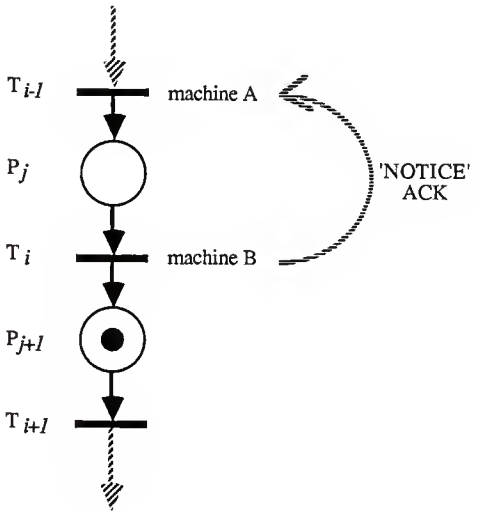


FIGURE 3.6

Example of Sending a 'NOTICE' ACK

The second situation occurs after the entire task has completed. In this case, a 'DONE' ACK is sent to the starting site and to any previous site if needed. 'Previous site' is taken to mean another site which sent a token to the current site and is waiting for the

current site to send an 'ACK' back. In Figure 3.7 after transition T*i* has completed, a 'DONE' ACK is sent to the starting site as well as to the previous site.

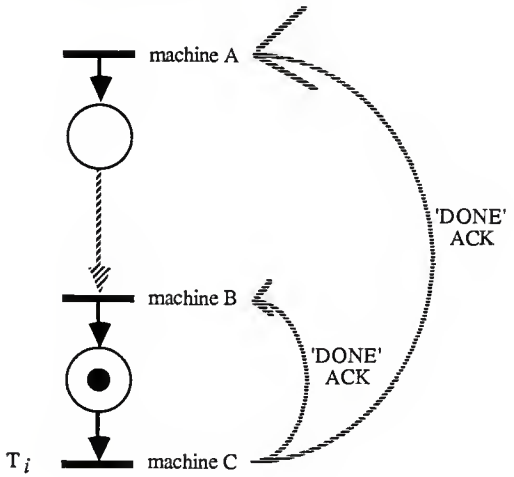


FIGURE 3.7

Example of Sending a 'DONE' ACK

A job that cannot continue constitutes the third situation. An example is a required function which does not exist in the distributed system. In this case, a 'DEAD' ACK is sent to the site where the job originated as well as to the previous site if needed. Assuming that transition T*i* of Figure 3.8 has failed and the agent on the current site has judged that the token cannot be further processed. Here, a 'DEAD' ACK will be sent to the starting site as well as to the previous site.

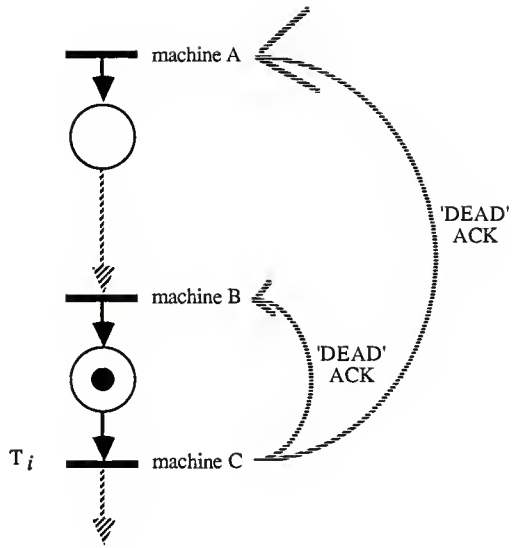


FIGURE 3.8

Example of Sending a 'DEAD' ACK

Since error recovery is one of the responsibilities of a CN agent, this system has high probability of recovering in the event of some machine(s) going down.

Let us consider the following situation where machine A sends a token to machine B so that the token can continue its processing. The agent on machine A will try to send the token to machine B several times. In case all of the attempts failed, the agent on A will re-route the token to some other site to perform the desired operation. Thus, a 'time out' must be defined for each transition. The time interval for the associated transition can be used to

judge how long the agent has to wait for an ACK before deciding to repeat or re-route the transition. The value for a 'time out' variable for a 'source' transition has to be declared large enough so that the maximum possible delay may occur before the starting site receives a 'DONE' message for a job. A copy of the token is always kept in the starting site until the site receives a 'DONE' ACK for that token. The worst case occurs when the source transition times out and has to restart the job all over again. This simple strategy guarantees some degree of recovery.

The CN can be hierarchically designed to handle more complicated jobs. A complex action associated with a transition can be thought of as a sub-CN, and a sub-CN can be treated in the same way as the highest level CN. In this way, recovery of the sub-CNs are made independent of each other. This simple recovery strategy can be applied from every sub-CN to the first level CN.

3.7 Summary

We discussed the design specification of this system with regard to the external CN, internal CN, intelligent token, and the CN agent. The external CN is designed for humans, while the internal CN is designed for machines. A token containing the marking of the CN, local data, and a routing history of the job is passed through the network. As a job is being processed, the CN agent at each site evaluates the job according to its marking and CN. The agent either performs the required action or forwards a token to the next processing site. Recovery problems are dealt with by a simple recovery strategy that is used in this prototype.

This system can be made more complete by designing a nice graphical interface between users and the system. In addition, transformation of the external CN to the internal CN should also be made automatic. Possible extensions of this work and concluding remarks will be given in the next chapter.

CHAPTER 4
CONCLUSION

4.1 Value of this work

Our system is designed to use the Petri net model to permit the distributed execution of a job. We have designed this system so that it is very easy to use, understand, and maintain. The internal construction of a job in this system is the result of a direct mapping by the user from its external representation. A CN agent is mainly responsible for overseeing that transitions are correctly carried out. The execution of jobs in this system is simple and straight-forward because the Petri net is a well-defined model.

A hierarchy of CNs may be necessary if a job gets complicated. This hierarchy makes the design of the control specification for a job easier. Actions of a simple job can be based upon the use of existing modules. A top-down hierarchy of CNs can help to solve a complicated job, complicated actions can be represented as subnets so that details of these actions could be hidden in lower levels of the CN. In this way, the program is easier to design and debug.

4.2 Extensions of this Work

One limitation of this work is the lack of a user interface. A graphical interface is needed to help guide the user during the process of designing the specification of a job. A good environment for graphic facility is needed for this purpose. A complete graphical interface package should not only provide the graphical environment, but also perform

some property checks based on the net created by the user. A graphical interface can also be used to simulate or track the actual movement of tokens in the net. Moreover, this graphical representation of a Petri net might also be automatically transformed into an internal data structure. Human mistakes can be eliminated by following such a procedure.

The ability to specify a number of sites where a merge transtion can be performed can be used to increase the reliablity of the system. Other extensions could be made depending upon the particular application of a job. For example, utilization of data files can be specified for transitions that either need the data from or output some data to particular data files.

4.3 Concluding Remarks

The intent of this work is to implement a model which can be applied to jobs that execute in a distributed environment. A 3B computer network was selected as the target environment of the prototype system since it is available in the Computing and Information Science Department at Kansas State University. Furthermore, this network offers a realistic environment for simulating our prototype system. An extended version of Petri nets is used because it provides the modeling power that is needed in a distributed system when parallelism is involved.

Bibliography

[ENCY83] Encyclopedia of Computer Science and Engineering, Van Nostrand Reinhold Company, 1983, pp. 563-565.

[FILM84] Filman R. E. and Friedman D. P., Coordinated Computing Tools and Techniques for Distributed Software, McGraw-Hill Book Company, 1984, 370 pages.

[KELL76] Keller, R. M. "Formal Verification of Parallel Programs," Communications of the ACM, Vol. 19, No. 7, July 1976, pp. 371-384.

[LAMP85] Lampson B. W. , Paul M. and Siegert H. J., Distributed Systems: architecture and implementation, Springer-Verlag, 1985, 510 pages.

[McBR83] McBride, R. A. and Unger, E. A. "Modeling Jobs In A Distributed System," ACM 0-89791-123-7/83/012/0032, 1983, pp. 32-41.

[McBR87] McBride, R. A. and Unger E. A. "The Representation and Distribution of Knowledge by a Petri Net," Draft Working Paper, Department of Computer Science, Kansas State University, March, 1987

[PETE77] Peterson, J. L. "Petri Nets" ACM Computing Surveys, Vol. 9, No. 3, Sept. 1977, pp. 223-252.

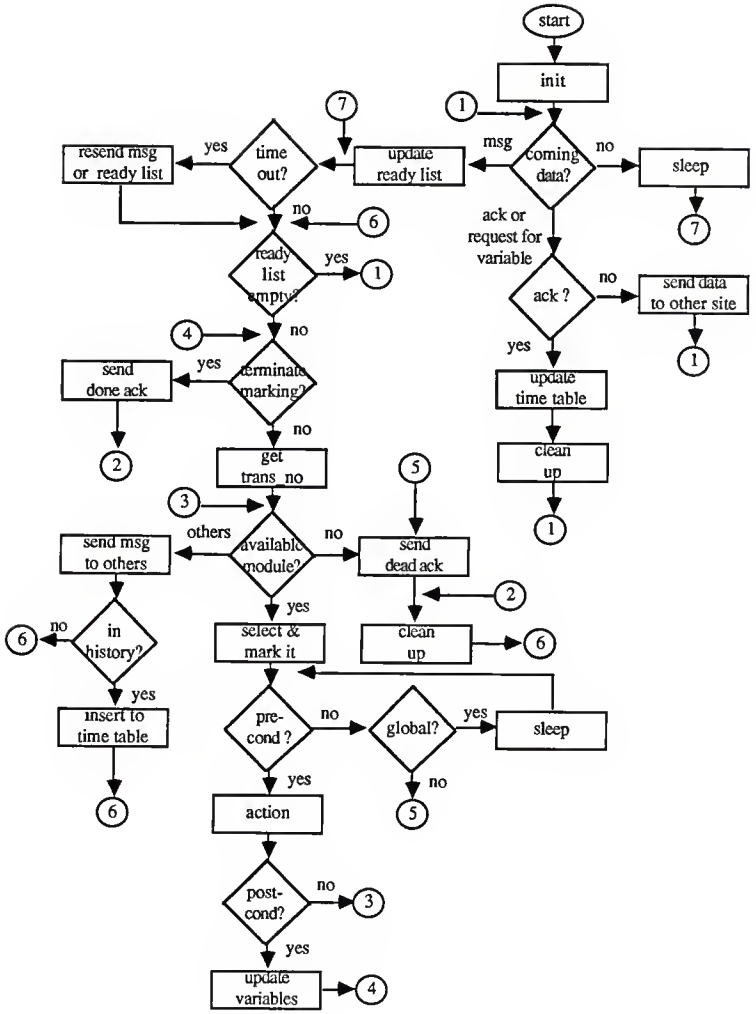
[PETE81] Peterson, J. L. Petri Net Theory and the Modeling of Systems, Prentice-Hall Inc., 1981, 290 pages

[SYMO80] Symons, F. J. W. Introduction to Numerical Petri Nets, a General Graphical Model of concurrent Processing Systems, Australian Telecommunication Research, Vol. 14, No. 1, 1980, pp. 28-32.

[TCP/IP] Enhanced TCP/IP WIN/3B User Guide.

Appendix A

Flowchart


start
init
1
coming data?
msg
no
sleep
7
ack or request for variable
ack ?
no
send data to other site
1
yes
update time table
clean up
1
7
update ready list
time out?
yes
resend msg or ready list
no
6
ready list empty?
yes
1
no
4
terminate marking?
yes
send done ack
2
no
get trans_no
5
send dead ack
2
clean up
6
3
available module?
others
no
send msg to others
yes
select & mark it
in history?
no
6
yes
insert to time table
6
pre-cond ?
no
global?
yes
sleep
yes
no
5
action
post-cond?
no
3
yes
update variables
4

Appendix B

Source Code Listing

```
#include <stdio.h>

#define TRACE 1
#define ACK 0141
#define DATA 0144
#define MSG 0155
#define NONE 0156
#define OTHER 0157
#define REQ 0162
#define WAIT 0167
#define YES 0171
#define REST 2
#define PATH "agent/"

#define DEAD 0
#define NOTICE 1
#define DONE 2

#define NEW 'n'
#define NO -1

#define TRUE 1
#define FALSE 0

#define MAX_MACHINE 7
#define MAX_PLACE1 6
#define MAX_TRANS1 6
#define MAX_PLACE2 9
#define MAX_TRANS2 8

#define MAX_IN 3
#define MAX_OUT 3
#define MAX_OP 2
#define MAX_IN_PLACE 3
#define MAX_OUT_PLACE 3
#define END -999
#define MAX_NODE_NAME 10
#define MAX_FILE_NAME 30
#define MAX_HISTORY 20
#define MAX_TIME_LEN 30
#define MAX_TIME_LIST 10
#define MASTER_COPY 1
#define MAX_RECORD 10
#define MAX_STACK 10
#define MAX_PRO_LEN 10
#define MAX_CMD 50
#define MAX_TRY 10
#define MARK 042
#define DEFAULT 5

typedef int BOOLEAN;

struct machine_type
{
```

```
    char op;            /* single operation symbol */
    char *node;         /* machine which support the op */
    char *process;      /* process' name */
};

struct record_type
{
    char to[MAX_NODE_NAME];  /* the destinataion node name */
    char process[MAX_PRO_LEN]; /* name of module */
};

struct wait_ack_data_type   /* waiting for ACK, DATA or variable */
{
    int id;                         /* msg or data id */
    char type;                      /* msg or data */
    char time[MAX_TIME_LEN];  /* last time stamp processed */
    int no_try;                     /* number of times retry the op */
    char wait_for;                  /* wait for ACK, DATA, or REQ */
    int interval;                   /* time interval before retry */
    struct record_type record[MAX_RECORD];  /* local history */
};

struct place_type
{
    char *merge_next;     /* site of a merge takes place */
    int merge_trans;      /* label of a merge transition */
};

struct action_type
{
    int ins[MAX_IN];  /* local input data, positive for a variable; negative for input plcae */
    char op;
};

struct trans_type
{
    char *take_p;                   /* executing site */
    int in_p[MAX_IN_PLACE];         /* input places */
    char *pre_cond;                 /* pre-condition */
    struct action_type act;         /* transition */
    char *post_cond;                /* post-condition */
    int out_p[MAX_OUT_PLACE];       /* output places */
    int time_out;                   /* time out interval */
};

struct petri_net_type    /* petri net includes places and transitions */
{
    struct place_type p[MAX_PLACE2];
    struct trans_type t[MAX_TRANS2];
};

struct history_type   /* history of the execution */
{
    int id;                         /* msg id */
```

```
    int trans;                          /* transition no */
    char op;                            /* opeartaion */
    char node[MAX_NODE_NAME];           /* op took place */
    char process[MAX_PRO_LEN];          /* process name,haven't used */
    char time[MAX_TIME_LEN];            /* time stamp of op */
};

struct data_type
{
    char site[MAX_NODE_NAME];           /* request site name */
    char name[2];                       /* variable name */
    int id;                             /* msg id */
};

struct messg_type
{
    int id;         /* message id - 4 digits: 1 -- type of CN; 2 -- instance number; 3,4 -- current place */
    char type;      /* MSG or DATA */
    int next_h;     /* index for next history location */
    char *start_node;                   /* the starts node */
    char *dest_node;                    /* the destination node */
    int marking[MAX_PLACE2];            /* marking */
    struct petri_net_type net;          /* petri net */
    int result;                         /* result */
    int inputs[2];                      /* local input data */
    struct history_type history[MAX_HISTORY];   /* routing history */
};

struct ack_type    /* acknowledgement */
{
    int id;         /* MSG or DATA's id */
    char type;      /* MSG or DATA */
    int info;       /* DEAD, NOTICE or DONE */
};

struct ready_type       /* ready list */
{
    int id;             /* msg id */
    char wait_for;      /* wait for ACK, DATA, or REQ otherwise is NEW */
};

char host[MAX_NODE_NAME];
int status;
int child;
```

```
/* ---------- in_record ---------- */
/* If the process has been tried then return TURE else return FALSE */

BOOLEAN in_record(process, p_record, op)
char *process;
struct record_type p_record[];
char op;
{
    int i;

    for (i = 0; i < 10 && p_record[i].to[0] != MARK && strcmp(p_record[i].process, process) != 0; i++)
        ;
    if (i >= 10 || p_record[i].to[0] == MARK)
        return(FALSE);
    else
        return(TRUE);
} /* in_record */


/* ---------- get_record ---------- */
/* place record into local history */

get_record(p_record, process, node)
struct record_type p_record[];
char *process, *node;
{
    int i;

    for (i = 0; i < 10 && p_record[i].to[0] != MARK; i++)
        ;
    if (i < 10)
    {
        strcpy(p_record[i].process, process);
        strcpy(p_record[i].to, node);
    }
} /* get_record */


/* ---------- available_module ---------- */
/* If there is an available module in its own site then return YES, if other site provides an available
module then reutrn OTHER, else return NONE */

char available_module(take_p, op, module_list, new_node, p_record, module)
char *take_p;
char op;
struct machine_type module_list[];
char new_node[];
struct record_type p_record[];
char *module;
{

    int i;
    BOOLEAN is_host = FALSE, got_one = FALSE;
    char found = NONE;
```

```
    for (i = 0; i < MAX_MACHINE && !is_host; i++)
    {
      if (op == module_list[i].op &&  !in_record(module_list[i].process, p_record, op))
       {
        if (!got_one)
        {
            if (strcmp(module_list[i].node, take_p) == 0 || strcmp("ANY", take_p) == 0)
            {
                *module = op;
                got_one = TRUE;
                if (strcmp(host, module_list[i].node) == 0)
                {
                    get_record(p_record,module_list[i].process,host);
                    is_host = TRUE;
                    found = YES;
                }
                else
                {
                    strcpy(new_node, module_list[i].node);
                    get_record(p_record,module_list[i].process,new_node);
                    found = OTHER;
                }
            }
        }
        else
            if (strcmp(host, module_list[i].node) == 0)
            {
                *module = op;
                get_record(p_record,module_list[i].process,host);
                is_host = TRUE;
                found = YES;
            }
      }
    }
    return(found);
} /* available_module */


/* ---------------------- coming_data ---------------------- */
/* receive coming information: message, ACK, or request */

char coming_data(msg_id, ack, req)
int *msg_id;
struct ack_type *ack;
struct data_type *req;
{
    int i;
    FILE *fp, *fopen();
    char file[MAX_FILE_NAME];

    for (i = 0; i < MAX_FILE_NAME; i++)
        file[i] = '\0';
    fp = NULL;
```

```
    for (i = 1100; i <= 3000 && fp == NULL; i++) /* open ack file which was sent to it */
    {
        sprintf(file, "a%s%d", host, i);
        fp = fopen(file, "r");
    }
    if (fp != NULL)
    {
        fscanf(fp, "%d", &ack->id);
        fscanf(fp, " %c", &ack->type);
        fscanf(fp, " %d", &ack->info);
        fclose(fp);
#if TRACE
printf("* received an ACK for message (%d)\n", ack->id);
#endif
        unlink(file);
        return(ACK);
    }
    for (i = 1100; i <= 3000 && fp == NULL; i++) /* open req file */
    {
        sprintf(file, "r%d", i);
        fp = fopen(file, "r");
    }
    if (fp != NULL)
    {
        fscanf(fp, "%s", req->site);
        fscanf(fp, " %s", req->name);
        fscanf(fp, " %d", &req->id);
        fclose(fp);
#if TRACE
prinif("* received an request for variable (%s) from site (%s) for message (%d)\n", req->name, req->site, req-
>id);
#endif
        unlink(file);
        return(REQ);
    }
    for (i = 1100; i <= 3000 && fp == NULL; i++) /* open msg file */
    {
        sprintf(file, "m%d", i);
        fp = fopen(file, "r");
    }
    if (fp != NULL)
    {
        fscanf(fp, "%d", msg_id);
        fclose(fp);
#if TRACE
printf("* received a MSG message (%d)\n", *msg_id);
#endif
        return(MSG);
    }
    else
        return(NONE);   /* no MSG or ACK files */
} /* coming data */
```

```
/* ---------- read_msg ---------- */
/* read in information */

BOOLEAN read_msg(type, id, msg)
char type;
int id;
struct messg_type *msg;
{
    int i, times = 0, max_place;
    FILE *fp, *fopen();
    char file[MAX_FILE_NAME], sys[MAX_CMD];

    sprintf(file, "%c%d", type, id);
    for(fp = NULL; fp == NULL && ++times < 3;)
        fp = fopen(file, "r");
    if (times >= 3 ) return(FALSE);
    fscanf(fp, "%d", &msg->id);
    fscanf(fp, " %c", &msg->type);
    fscanf(fp, " %d", &msg->next_h);

    if (id/1000 == 1)
        max_place = MAX_PLACE1;
    else
        max_place = MAX_PLACE2;
    for (i = 0; i < max_place; i++)   /* marking */
        fscanf(fp, "%1d", &msg->marking[i]);

    fscanf(fp, "%d", &msg->result);
    fscanf(fp, "%d", &msg->inputs[0]);
    fscanf(fp, "%d", &msg->inputs[1]);

    if (msg->next_h > 0)
    {
        fscanf(fp, "\n");
        for (i = 0; i < msg->next_h; i++) /* history */
        {
            fscanf(fp, "%d", &msg->history[i].id);
            fscanf(fp, " %d", &msg->history[i].trans);
            fscanf(fp, " %c", &msg->history[i].op);
            fscanf(fp, " %s", msg->history[i].node);
            fscanf(fp, " %s\n", msg->history[i].time);
        }
    }
    else /* treat it as the start node */
    {
        strcpy(msg->history[0].node, host);
        msg->history[0].id = msg->id;
        msg->history[0].trans = 0;
        msg->history[0].op = 's';
        strcpy(msg->history[0].time, get_time());
        msg->next_h++;
    }
    fclose(fp);
```

```
    /* after read the msg move the msg to wait file */
    if (type == MSG)
    {
        sprintf(sys, "mv %s w%d", file, id);
        system(sys);
    }
    return(TRUE);
} /* read_msg */


/* ---------- execute_action ---------- */
/* execute a transition by creating a child process to perform the function */

BOOLEAN execute_action(var, trans_no, module, t_m)
struct data_type var[];
int trans_no;
char module;
struct messg_type *t_m;
{
    int d1, d2, d3, i;
    FILE *fp, *fopen();
    char sys[MAX_CMD], *function;

    if (module == 'f')
    {
        fork_it(trans_no, t_m);
        return(TRUE);
    }
    else
    {
        d1 = t_m->inputs[0];
        d2 = t_m->inputs[1];
        switch(module)
        {
            case '+':
                function = "plus";
                break;
            case '-':
                function = "minus";
                break;
            case '*':
                function = "multi";
                break;
            case '/':
                function = "divide";
                break;
            default: break;
        }
        sprintf(sys,"%s %d %d %s",function,d1,d2,"temp");
        system(sys);

        for (i = 1, fp = NULL; i <= 2 && fp == NULL; ++i, sleep(2))
            fp = fopen("temp", "r");
        if (fp != NULL)
```

```
        {
            fscanf(fp, "%d", &d3);
            fclose(fp);
            unlink("temp");
            t_m->result = d3;
            return(TRUE);
        }
        else
            return(FALSE);
    }
} /* execute_action */


/* ---------- count_out_place ---------- */
/* return the number of out put places of a transition */

count_out_place(trans_no, t_m)
int trans_no;
struct messg_type *t_m;
{
    int i;

    for (i = 0; i < MAX_OUT_PLACE && t_m->net.t[trans_no].out_p[i] != END; i++)
        ;
    return(i);
}


/* ---------- fork_it ---------- */
/* perform a fork transition of a Petri net, the marking will be different for every sibling token */

fork_it(trans_no, t_m)
int trans_no;
struct messg_type *t_m;
{
    int pid1, pid2, no_child, num, place_no = 0, total, t, i;
    char temp[4], file[MAX_FILE_NAME], f1[6], f2[6];

    for (i = 0, num = t_m->net.t[trans_no].in_p[i]; num != END;)
    {
        t_m->marking[num] = 0;
        num = t_m->net.t[trans_no].in_p[++i];
    }
    total = count_out_place(trans_no, t_m);
    fflush(stdout);
    if ((pid1 = fork()) != 0)
    {
        t_m->id = t_m->id /100 * 100 + t_m->net.t[trans_no].out_p[place_no];
        num = t_m->net.t[trans_no].out_p[place_no];
        t_m->marking[num] = 1;
        sprintf(f2, "w%d", t_m->id);
        write_msg_to_file(t_m, f2);
    }
    else
```

```
    {           /* create a new msg id for child */
        child = getpid();
        t_m->id = t_m->id /100 * 100 + t_m->net.t[trans_no].out_p[++place_no];
        num = t_m->net.t[trans_no].out_p[place_no];
        t_m->marking[num] = 1;
        t_m->next_h = 0; /* wipe out the history */
        /* write marking and data to parent */
        sprintf(f1, "w%d", t_m->id);
        write_msg_to_file(t_m, f1);
        if (total > 2 && (pid2 = fork()) == 0)
        {           /* create another msg id for child */
            child = getpid();
            t_m->marking[num] = 0;
            t_m->id = t_m->id /100 * 100 + t_m->net.t[trans_no].out_p[++place_no];
            num = t_m->net.t[trans_no].out_p[place_no];
            t_m->marking[num] = 1;
            sprintf(f1, "w%d", t_m->id);
            write_msg_to_file(t_m, f1);
        }
    }
}


/* ---------------------  pre_trans_no  --------------------- */
/* return the previous transition label */

int pre_trans_no(msg)
struct messg_type *msg;
{
    int i, max_trans;
    BOOLEAN get_it = FALSE;

    if (msg->id/100 == 1) max_trans = MAX_TRANS1;
    else max_trans = MAX_TRANS2;
    for (i = 1; i < max_trans; i++)
    {
        if(msg->marking[msg->net.t[i].out_p[0]])
        {
            get_it == TRUE;
            return(i);
        }
    }
    return(-1);
} /* pre_trans_no */

/* ---------------------  get_trans_no  --------------------- */
/* check the marking to find out current trans no */

int get_trans_no(msg)
struct messg_type *msg;
{
    int i, j, max_trans;
    BOOLEAN get_it = TRUE;
```

```
    if (msg->id/100 == 1) max_trans = MAX_TRANS1;
    else max_trans = MAX_TRANS2;
    for (i = 1; i < max_trans; i++, get_it = TRUE)
    {
        for (j = 0; j < MAX_IN_PLACE && msg->net.t[i].in_p[j] != END; j++)
            if (!msg->marking[msg->net.t[i].in_p[j]])
            {
                get_it = FALSE;
                break;
            }
        if (get_it == TRUE)
        {
            return(i);
        }
    }
    return(-1);
} /* get_trans_no */


/* ------------------  get_host  ------------------ */
/* get host node name */

char *get_host()
{
    char name[MAX_NODE_NAME], temp[MAX_CMD], file[MAX_FILE_NAME];
    FILE *fopen(), *fp;

    strcpy(file, "hostname1");
    sprintf(temp, "hostname > %s", file);
    system(temp);
    fp = fopen(file, "r");
    fscanf(fp, "%s", name);
    fclose(fp);
    unlink(file);
    return(name);
} /* get_host */


/* ------------------  get_val  ------------------ */
/* get the local input variable */

BOOLEAN get_val(trans_no, msg, var, wait_for)
int trans_no;
struct messg_type *msg;
struct data_type var[];
char *wait_for;
{
    int index1, index2, times, t1;
    char var1[2], var2[2], sys[MAX_CMD], f1[10];
    FILE *fp, *fopen();
    struct messg_type *tmp;

    index1 = msg->net.t[trans_no].act.ins[0];
    index2 = msg->net.t[trans_no].act.ins[1];
```

```
var1[0] = index1 + 0101; var1[1] = '\0';
var2[0] = index2 + 0101; var2[1] = '\0';
fp = NULL;

if (index1 == END) return(TRUE);
if (index1 >= 0 && msg->inputs[0] == -999) /* from variable list */
{
    if (strcmp(host, var[index1].site) != 0) /* var is not at host */
    {
        sprintf(f1, "%s%d", var1, msg->id);
        fp = fopen(f1, "r");
        if (fp == NULL)
            remote_get_val(index1, var, var1, msg);
        else
        {
            fscanf(fp, "%d", &msg->inputs[0]);
            fclose(fp);
            unlink(f1);
        }
    }
    else /* var is at host */
    {
        fp = fopen(var1, "r");
        if (fp != NULL)
        {
            fscanf(fp, "%d", &msg->inputs[0]);
            fclose(fp);
        }
    }
    fclose(fp);
    if (fp == NULL)
    {
        *wait_for = REQ;
        return(FALSE);
    }
} /* else from input places */
else
{
    if (msg->inputs[0] == -999)
        msg->inputs[0] = msg->result;
}
if (index2 >= 0 && msg->inputs[1] == -999)  /* from variable list */
{
    if (strcmp(host, var[index2].site) != 0)     /* var is not at host */
    {
        sprintf(f1, "%s%d", var2, msg->id);
        fp = fopen(f1, "r");
        if (fp == NULL)
            remote_get_val(index2, var, var2, msg);
        else
        {
            fscanf(fp, "%d", &msg->inputs[1]);
            fclose(fp);
            unlink(f1);
```

```
            }
        }
        else /* var is at host */
        {
            fp = fopen(var2, "r");
            if (fp != NULL)
            {
                fscanf(fp, "%d", &msg->inputs[1]);
                fclose(fp);
            }
        }
        fclose(fp);
        if (fp == NULL)
        {
            *wait_for = REQ;
            return(FALSE);
        }
        else
            return(TRUE);
    }
    else /* else from input places */
    {
        if (msg->inputs[1] == -999)
        {
            t1 = msg->id/100*100 + msg->net.t[trans_no].in_p[1];
            for(fp = NULL, times = 1; fp == NULL && ++times <= 3;)
            {
                sprintf(f1, "d%d", t1);
                fp = fopen(f1, "r");
                if (fp == NULL) sleep(3);
            }
            fclose(fp);
            /* before real merge action, we have to update the marking first, up to this point, only
                have to know the marking of the child */
            if (times <= 3)
            {
                tmp = (struct messg_type *) malloc(sizeof(struct messg_type));
                init_msg(tmp, msg->id/1000);
                read_msg(DATA, t1, tmp);
                combine_info(msg, tmp);
                free(tmp);
                return(TRUE);
            }
            else
            {
                *wait_for = DATA;
                return(FALSE);
            }

        }
    }
} /* get_val */
```

```
/* ------------------  remote_get_val  ------------------ */
/* request variable from other site */

remote_get_val(index, var, variable, msg)
int index;
struct data_type var[];
char variable[];
struct messg_type *msg;
{
    char sys[MAX_CMD], file[MAX_FILE_NAME];
    FILE *fopen(), *fp;
    int times;

    sprintf(file, "r%d", msg->id);
    fp = fopen(file, "w");
    fprintf(fp, "%s", host);
    fprintf(fp, " %s", variable);
    fprintf(fp, " %d", msg->id);
    fclose(fp);
    sprintf(sys, "rcp %s %s:%s", file, var[index].site, file);
    system(sys);
    sprintf(sys, "rm %s", file);
#if TRACE
printf("request variable (%s) from other site (%s) for message (%d)\n", variable, var[index].site, msg->id);
#endif
    system(sys);
} /* remote_get_val */


/* ------------------  init  ------------------ */
/* initialization */

init(var, ready_stack, wait_ack_data, module_list)
struct data_type var[];
struct ready_type ready_stack[];
struct wait_ack_data_type wait_ack_data[];
struct machine_type module_list[];
{
    int i;

    init_var(var);
    strcpy(host, get_host());
    for (i = 0; i < MAX_TIME_LIST; i++)
        wait_ack_data[i].id = END;
    ready_stack[0].id = 1;
    init_machine(module_list);
} /* init */


/* ------------------  init_var  ------------------ */
/* assume all varialbes are in site november */

init_var(var)
struct data_type var[];
```

```
{
    int i;
    char name;

    for (name = 'A', i = 0; i < 10; i++)
    {
        var[i].name[0] = name++;
        var[i].name[1] = '\0';
        strcpy(var[i].site, "november");
    }
}


/* ---------------- init_machine ---------------- */
/* initialize machine list */

init_machine(module_list)
struct machine_type module_list[];
{
    module_list[0].op = '+';
    module_list[0].node = "november";
    module_list[0].process = "plus1";

    module_list[1].op = '*';
    module_list[1].node = "november";
    module_list[1].process = "times1";

    module_list[2].op = '-';
    module_list[2].node = "hotel";
    module_list[2].process = "minus1";

    module_list[3].op = '-';
    module_list[3].node = "hotel";
    module_list[3].process = "minus2";

    module_list[4].op = 'f';
    module_list[4].node = "november";
    module_list[4].process = "fork1";

    module_list[5].op = 'f';
    module_list[5].node = "hotel";
    module_list[5].process = "fork2";

    module_list[6].op = '/';
    module_list[6].node = "november";
    module_list[6].process = "divide1";
} /* init_machine */
```

```
/* ---------------     init_msg     ---------------- */
/* initialize the static structure of Petri net for messages */

init_msg(msg, msg_index)
struct messg_type *msg;
int msg_index;
{
   if (msg_index == 1)
   {
     msg->start_node = "november";
     msg->dest_node = "november";

     msg->net.p[0].merge_next = "NO";
     msg->net.p[0].merge_trans = -1;
     msg->net.p[1].merge_next = "NO";
     msg->net.p[1].merge_trans = -1;
     msg->net.p[2].merge_next = "NO";
     msg->net.p[2].merge_trans = -1;
     msg->net.p[3].merge_next = "november";
     msg->net.p[3].merge_trans = 4;
     msg->net.p[4].merge_next = "november";
     msg->net.p[4].merge_trans = 4;
     msg->net.p[5].merge_next = "NO";
     msg->net.p[5].merge_trans = -1;

     msg->net.t[0].in_p[0] = END;
     msg->net.t[0].out_p[0] = 0;
     msg->net.t[0].out_p[1] = END;
     msg->net.t[0].act.ins[0] = END;
     msg->net.t[0].act.op = 's';
     msg->net.t[0].take_p= "ANY";
     msg->net.t[0].time_out = DEFAULT + 10;

     msg->net.t[1].in_p[0] = 0;
     msg->net.t[1].in_p[1] = END;
     msg->net.t[1].out_p[0] = 1;
     msg->net.t[1].out_p[1] = 2;
     msg->net.t[1].out_p[2] = END;
     msg->net.t[1].act.ins[0] = END;
     msg->net.t[1].act.op = 'f';
     msg->net.t[1].take_p= "ANY";
     msg->net.t[1].time_out = DEFAULT;

     msg->net.t[2].in_p[0] = 1;
     msg->net.t[2].in_p[1] = END;
     msg->net.t[2].out_p[0] = 3;
     msg->net.t[2].out_p[1] = END;
     msg->net.t[2].act.ins[0] = 0;
     msg->net.t[2].act.ins[1] = 1;
     msg->net.t[2].act.ins[2] = END;
     msg->net.t[2].act.op = '+';
     msg->net.t[2].take_p= "ANY";
     msg->net.t[2].time_out = DEFAULT;
```

```
  msg->net.t[3].in_p[0] = 2;
  msg->net.t[3].in_p[1] = END;
  msg->net.t[3].out_p[0] = 4;
  msg->net.t[3].out_p[1] = END;
  msg->net.t[3].act.ins[0] = 2;
  msg->net.t[3].act.ins[1] = 3;
  msg->net.t[3].act.ins[2] = END;
  msg->net.t[3].act.op = '-';
  msg->net.t[3].take_p= "ANY";
  msg->net.t[3].time_out = DEFAULT;

  msg->net.t[4].in_p[0] = 3;
  msg->net.t[4].in_p[1] = 4;
  msg->net.t[4].in_p[2] = END;
  msg->net.t[4].out_p[0] = 5;
  msg->net.t[4].out_p[1] = END;
  msg->net.t[4].act.ins[0] = -3;
  msg->net.t[4].act.ins[1] = -4;
  msg->net.t[4].act.ins[2] = END;
  msg->net.t[4].act.op = '*';
  msg->net.t[4].take_p= "november";
  msg->net.t[4].time_out = DEFAULT;
} /* msg_index == 1 */

if (msg_index == 2)
{
  msg->start_node = "november";
  msg->dest_node = "november";

  msg->net.p[0].merge_next = "NO";
  msg->net.p[0].merge_trans = -1;
  msg->net.p[1].merge_next = "NO";
  msg->net.p[1].merge_trans = -1;
  msg->net.p[2].merge_next = "NO";
  msg->net.p[2].merge_trans = -1;
  msg->net.p[3].merge_next = "NO";
  msg->net.p[3].merge_trans = -1;
  msg->net.p[4].merge_next = "november";
  msg->net.p[4].merge_trans = 5;
  msg->net.p[5].merge_next = "november";
  msg->net.p[5].merge_trans = 5;
  msg->net.p[6].merge_next = "november";
  msg->net.p[6].merge_trans = 6;
  msg->net.p[7].merge_next = "november";
  msg->net.p[7].merge_trans = 6;
  msg->net.p[8].merge_next = "NO";
  msg->net.p[8].merge_trans = -1;

  msg->net.t[0].in_p[0] = END;
  msg->net.t[0].out_p[0] = 0;
  msg->net.t[0].out_p[1] = END;
  msg->net.t[0].act.ins[0] = END;
  msg->net.t[0].act.op = 's';
  msg->net.t[0].take_p= "ANY";
```

```
msg->net.t[0].time_out = DEFAULT + 20;

msg->net.t[1].in_p[0] = 0;
msg->net.t[1].in_p[1] = END;
msg->net.t[1].out_p[0] = 1;
msg->net.t[1].out_p[1] = 2;
msg->net.t[1].out_p[2] = 3;
msg->net.t[1].out_p[3] = END;
msg->net.t[1].act.ins[0] = END;
msg->net.t[1].act.op = 'f';
msg->net.t[1].take_p= "ANY";
msg->net.t[1].time_out = DEFAULT;

msg->net.t[2].in_p[0] = 1;
msg->net.t[2].in_p[1] = END;
msg->net.t[2].out_p[0] = 4;
msg->net.t[2].out_p[1] = END;
msg->net.t[2].act.ins[0] = 0;
msg->net.t[2].act.ins[1] = 1;
msg->net.t[2].act.ins[2] = END;
msg->net.t[2].act.op = '+';
msg->net.t[2].take_p= "ANY";
msg->net.t[2].time_out = DEFAULT;

msg->net.t[3].in_p[0] = 2;
msg->net.t[3].in_p[1] = END;
msg->net.t[3].out_p[0] = 5;
msg->net.t[3].out_p[1] = END;
msg->net.t[3].act.ins[0] = 2;
msg->net.t[3].act.ins[1] = 3;
msg->net.t[3].act.ins[2] = END;
msg->net.t[3].act.op = '-';
msg->net.t[3].take_p= "ANY";
msg->net.t[3].time_out = DEFAULT;

msg->net.t[4].in_p[0] = 3;
msg->net.t[4].in_p[1] = END;
msg->net.t[4].out_p[0] = 6;
msg->net.t[4].out_p[1] = END;
msg->net.t[4].act.ins[0] = 4;
msg->net.t[4].act.ins[1] = 5;
msg->net.t[4].act.ins[2] = END;
msg->net.t[4].act.op = '-';
msg->net.t[4].take_p= "ANY";
msg->net.t[4].time_out = DEFAULT;

msg->net.t[5].in_p[0] = 4;
msg->net.t[5].in_p[1] = 5;
msg->net.t[5].in_p[2] = END;
msg->net.t[5].out_p[0] = 7;
msg->net.t[5].out_p[1] = END;
msg->net.t[5].act.ins[0] = -4;
msg->net.t[5].act.ins[1] = -5;
msg->net.t[5].act.ins[2] = END;
```

```
    msg->net.t[5].act.op = '*';
    msg->net.t[5].take_p= "november";
    msg->net.t[5].time_out = DEFAULT;

    msg->net.t[6].in_p[0] = 7;
    msg->net.t[6].in_p[1] = 6;
    msg->net.t[6].in_p[2] = END;
    msg->net.t[6].out_p[0] = 8;
    msg->net.t[6].out_p[1] = END;
    msg->net.t[6].act.ins[0] = -7;
    msg->net.t[6].act.ins[1] = -6;
    msg->net.t[6].act.ins[2] = END;
    msg->net.t[6].act.op = '/';
    msg->net.t[6].take_p= "november";
    msg->net.t[6].time_out = DEFAULT;
  } /* msg_index == 2 */
} /* init_msg */


/* ------------------    main    ---------------------- */
/* main body */

main()
{
    struct machine_type module_list[MAX_MACHINE];
    struct wait_ack_data_type wait_a_d[MAX_TIME_LIST];
    struct ack_type *ack;
    char *malloc(), *free(), *temp, ans, sys[MAX_CMD];
    struct ready_type ready_stack[MAX_STACK];
    int msg_id, test = 30, i, pid;
    struct record_type p_record[MAX_RECORD];
    struct data_type var[10], *req;

    ack = (struct ack_type *) malloc(sizeof(struct ack_type));
    req = (struct data_type *) malloc(sizeof(struct data_type));
    init(var, ready_stack, wait_a_d, module_list);
    for (; test-- > 0;)
        switch(ans = coming_data(&msg_id, ack, req))
        {
            case ACK:
                delete_wait_a_d(wait_a_d, ack);
                break;
            case REQ:
                sprintf(sys, "rcp %s %s:%s%d", req->name, req->site, req->name, req->id);
                system(sys);
                break;
            case NONE:
                sleep(REST);
            case MSG:
                /* keep the new msg in ready_stack, check the time out event first */
                if (ans == MSG)
                {
                    ready_stack[ready_stack[0].id].wait_for = NEW;
                    ready_stack[ready_stack[0].id++].id = msg_id;
```

```
                    }
                    check_time_out(wait_a_d, ready_stack);
                    for (i = ready_stack[0].id-1; i > 0; --i)
                    {
#if TRACE
printf("> process message (%d)\n", ready_stack[i].id);
#endif
                        process_msg(var,ready_stack[i].id,module_list,wait_a_d,p_record,ready_stack[i].wait_
for);
                    }
                    ready_stack[0].id = 1;
                    break;
                default:
                    break;
            } /* switch */
        free(ack);
        free(req);
} /* main */


/* ------------------  in_wait  ---------------------- */
/* if message is kept in waiting list then return TRUE else return FASLE */

BOOLEAN in_wait(id, wait_a_d, index)
int id;
struct wait_ack_data_type wait_a_d[];
int *index;
{
    int i;

    for (i = 0; i < MAX_TIME_LIST; i++)
    {
        if(wait_a_d[i].id == id)
        {
            *index = i;
            return(TRUE);
        }
    }
    *index = -1;
    return(FALSE);
} /* in_wait */


/* ------------------  combine_wait ---------------------- */
/* combine parent and child's waiting list */

combine_wait(w)
struct wait_ack_data_type w[];
{
    FILE *fp, *fopen();
    int i, j, r1 = 1, r2 = 1;
    struct wait_ack_data_type *t;

    if (child != 0)
```

```
    {
        fp = fopen("wait", "a");
        for (i = 0; i < 10; i++)
            if (w[i].id != END)
            {
                fprintf(fp,"%d %c %s %d %c %d",
                    w[i].id,w[i].type,w[i].time,w[i].no_try,w[i].wait_for,w[i].interval);
                for (j = 0; j < 10; j++)
                    fprintf(fp," %s %s",w[i].record[j].to,w[i].record[j].process);
                fprintf(fp, "\n");
            }
        fclose(fp);
    }
    else
    {
        fp = fopen("wait", "r");
        if (fp > 0) /* parent read in the wait list and combine */
        {
            t = (struct wait_ack_data_type *) malloc(sizeof(struct wait_ack_data_type));
            for (i = 0; i < 10 && r1 > 0; i++)
            {
                r1=fscanf(fp,"%d %c %s %d %c %d",&t->id,&t->type,t->time,&t->no_try,
                    &t->wait_for,&t->interval);
                for (j = 0; j < 10 && r1 > 0 && r2 > 0; j++)
                    r2=fscanf(fp," %s %s",t->record[j].to,t->record[j].process);
                fscanf(fp, "\n");
                /* combine the t into the wait list */
                for (j = 0; j < 10 && r1 > 0 && w[j].id != t->id; j++)
                    ;
                if ( j >= 10 )
                    for (j = 0; j < 10 && w[j].id != END && r1 > 0; j++)
                        ;
                if ( j < 10 && r1 > 0)
                {
                    w[j].id = t->id;
                    w[j].type = t->type;
                    strcpy(w[j].time, t->time);
                    w[j].no_try = t->no_try;
                    w[j].wait_for = t->wait_for;
                    w[j].interval = t->interval;
                    dump_record(w[j].record,t->record);
                }
            }
            fclose(fp);
            unlink("wait");
            free(t);
        }
    }
} /* combine_wait */


/* ---------------  process_msg  ---------------------- */
/* process the message */
```

```
process_msg(var, id, module_list, wait_a_d, p_record, wait_for)
struct data_type var[];
int id;
struct machine_type module_list[];
struct wait_ack_data_type wait_a_d[];
struct record_type p_record[];
char wait_for;
{
    struct messg_type *msg, *t_m;
    struct ack_type *ack;
    char to[MAX_NODE_NAME], op, module, temp[MAX_FILE_NAME];
    BOOLEAN is_msg = FALSE, post_cond = TRUE, done, why;
    char resp;
    int i, trans_no, index, max_place;

    msg = (struct messg_type *) malloc(sizeof(struct messg_type));
    t_m = (struct messg_type *) malloc(sizeof(struct messg_type));
    ack = (struct ack_type *) malloc(sizeof(struct ack_type));

    /* if the msg is from wait list then the local history dump back to p_record else re-init p_record */
    if (!in_wait(id, wait_a_d, &index))
        for (i = 0; i < MAX_RECORD; i++)
        {
            p_record[i].to[0] = MARK;
            p_record[i].to[1] = '\0';
            p_record[i].process[0] = MARK;
            p_record[i].process[1] = '\0';
        }
    else
        dump_record(p_record, wait_a_d[index].record);
    child = 0;
    init_msg(msg, id/1000);
    if (wait_for == NEW)
        why=read_msg(MSG, id, msg);
    else
        why=read_msg(WAIT, id, msg);
    do {
        if (id/1000 == 1) max_place = MAX_PLACE1;
        else max_place = MAX_PLACE2;
        if (msg->marking[max_place-1])
        {
            ack->id = msg->id;
            ack->type = MSG;
            ack->info = DONE;
#if TRACE
printf("msg id=(%d) is DONE\n", ack->id/100*100);
#endif
            delete_wait_a_d(wait_a_d, ack);
            is_msg = FALSE;
        }
        else
        {
            if (post_cond)
            {
```

```
                trans_no = get_trans_no(msg);
                op = msg->net.t[trans_no].act.op;
            }
            switch(available_module(
                msg->net.t[trans_no].take_p,
                op, module_list, to, p_record, &module))
            {
                case NONE:
                    ack->id = msg->history[0].id; ack->type = MSG;
                    ack->info = DEAD;
                    if (strcmp(msg->history[msg->next_h-1].node,host) != 0)
                        send_ack(ack,msg->history[msg->next_h-1].node);
#if TRACE
printf("There is a dead msg. pid=(%d),op=(%c),t=(%d)\n",getpid(),op,trans_no);
#endif
                    delete_wait_a_d(wait_a_d, ack);
                    is_msg = FALSE;
                    break;
                case OTHER:
                    sprintf(temp, "w%d", msg->id);
                    write_msg_to_file(msg, temp);
                    send_msg(MSG, msg->id, to);
                    if (strcmp(msg->history[msg->next_h-1].node, host) == 0)
                    {
                        insert_wait_a_d(to,wait_a_d,msg->id,msg->type,p_record,ACK,
                            msg->net.t[trans_no].time_out);
                    }
                    else /* current node will not be in the history */
                    {
                        sprintf(temp, "w%d", msg->id);
                        unlink(temp);  /* remove the wfile */
                    }
                    is_msg = FALSE;
                    break;
                case YES:
                    /* specify the pre_cond to be the time problem */
                    /* while (!pre_cond) sleep(2); */
                    if (msg->marking[0])
                    {
#if TRACE
printf("insert the msg (%d) for the source transiton to wait list\n", msg->id);
#endif
                        insert_wait_a_d("none",wait_a_d,msg->id,msg->type,p_record,ACK,
                            msg->net.t[trans_no].time_out);
                    }
                    init_msg(t_m, id/1000);
                    copy_msg(t_m, msg);
                    if (!get_val(trans_no, t_m, var,&wait_for))
                    {
                        is_msg = FALSE;
                    for (done = FALSE, i = 0; i < MAX_RECORD && !done; ++i)
                    {
                        if (p_record[i].to[0] == MARK)
                            done = TRUE;
```

```
                    }
                    i = i - 2;
                    p_record[i].to[0] = MARK; p_record[i].to[1] = '\0';
#if TRACE
printf("insert message (%d) into wait list for (%c)\n",msg->id ,wait_for);
#endif
                        insert_wait_a_d("NO",wait_a_d,msg->id,msg->type,p_record,wait_for,
                            msg->net.t[trans_no].time_out);
                        break;
                    }
                    if (!execute_action(var, trans_no, module, t_m))
                    {
                        is_msg = FALSE;
                        break;
                    }
                    if (post_cond)
                    {
                        /* o-ok, c-child, m-msg, f-failed */
                        resp = 'o';
                        if (in_wait(t_m->id, wait_a_d, &index) && (wait_for == DATA ||
                            wait_for == REQ))
                            wait_a_d[index].id = END;
                        update_msg(trans_no,t_m,msg,&resp,var);
                        strcpy(temp,msg->net.p[msg->net.t[trans_no].out_p[0]].merge_next);
                        if (resp == 'c' || resp == 'm')
                        {
                            is_msg = FALSE;
                            if (strcmp(temp, host) != 0)
                            {
#if TRACE
printf("insert data message (%d) into wait list, wait for ACK\n", msg->id);
#endif
                                insert_wait_a_d("NO",wait_a_d,msg->id,DATA,p_record,ACK,
                                    msg->net.t[trans_no].time_out);
                            }
                        } else
                        if (resp == 'f')
                        {   /* erase the last history */
                            for (done = FALSE, i = 0; i < MAX_RECORD && !done; ++i)
                            {
                                if (p_record[i].to[0] == MARK)
                                    done = TRUE;
                            }
                            i = i - 2;
                            p_record[i].to[0] = MARK; p_record[i].to[1] = '\0';
                            is_msg = FALSE;
#if TRACE
printf("insert message (%d) into wait list, wait for DATA\n", msg->id);
#endif
                            insert_wait_a_d("NO",wait_a_d,msg->id,DATA,p_record,DATA,
                                msg->net.t[trans_no].time_out);
                        }
                        else /* resp == 'o' */
                            is_msg = TRUE;
```

```
                        }
                        break;
                    default: break;
                } /* switch */
            } /* else */
        } while (is_msg); /* do */
        combine_wait(wait_a_d);
        free(msg); free(t_m); free(ack);
        if (child != 0) exit();
} /* process msg */
```

```
/* ---------------    write_msg_to_file    ---------------- */
/* write message to a file */

write_msg_to_file(msg, file)
struct messg_type *msg;
char file[];
{
    FILE *fp, *fopen();
    int i, max_place;

    fp = fopen(file, "w");
    fprintf(fp, "%d %c %d ", msg->id, msg->type, msg->next_h);
    if (msg->id/1000 == 1)
        max_place = MAX_PLACE1;
    else
        max_place = MAX_PLACE2;
    for (i = 0; i < max_place; i++)
        fprintf(fp, "%d", msg->marking[i]);
    fprintf(fp, " %d", msg->result);
    fprintf(fp, " %d", msg->inputs[0]);
    fprintf(fp, " %d", msg->inputs[1]);
    fprintf(fp, "\n");
    for (i = 0; i < msg->next_h; i++)
    {
        fprintf(fp, "%d", msg->history[i].id);
        fprintf(fp, " %d", msg->history[i].trans);
        fprintf(fp, " %c", msg->history[i].op);
        fprintf(fp, " %s", msg->history[i].node);
        fprintf(fp, " %s\n", msg->history[i].time);
    }
    fclose(fp);
} /* write_msg_to_file */


/* ----------------    send_ack    ---------------- */
/* send ACK to other site */

send_ack(ack, node)
struct ack_type *ack;
char node[];
{
    char f1[MAX_FILE_NAME], sys[MAX_CMD];
    FILE *fp, *fopen();

    sprintf(f1, "a%s%d", node, ack->id);
    fp = fopen(f1, "w");
    fprintf(fp, "%d %c %d", ack->id, ack->type, ack->info);
    fclose(fp);
#if TRACE
printf("send ack to site (%s) for message (%d)\n", node, ack->id);
#endif
    sprintf(sys,"rcp %s %s:%s", f1, node, f1);
    system(sys);
    unlink(f1);
```

```
} /* send_ack */


/* ------------------  send_ack_cleanup  ------------------ */
/* send ack mainly after merge transition */

send_ack_cleanup(msg)
struct messg_type *msg;
{
    char tmp[MAX_CMD], place_no[2];
    int pre_trans, no, place, i;
    struct ack_type *ack;

    ack = (struct ack_type *) malloc(sizeof(struct ack_type));
    no = msg->next_h - 2;
    if (no >= 0) pre_trans = pre_trans_no(msg); /* get merge trans no */
    if (no >= 0 && strcmp(msg->history[no].node, host) != 0)/* from child */
    {
        if (strcmp(msg->net.t[pre_trans].take_p, msg->history[no].node) != 0
            && strcmp("ANY",msg->net.t[pre_trans].take_p) != 0)
        {           /* right after merge transaction */
            ack->id = msg->id/100*100+msg->net.t[pre_trans].in_p[1];
            ack->type = DATA;
            ack->info = NOTICE;
            send_ack(ack, msg->history[no].node);
        }
        else        /* general sending ack */
        {
            ack->id = msg->history[0].id;
            ack->type = MSG;
            ack->info = NOTICE;
            send_ack(ack, msg->history[no].node);
        }
    }
}
/* send_ack_cleanup */


/* ------------------  send_old_msg  ------------------ */
/* resend old msg or data file */

send_old_msg(i, wait_a_d)
int i;
struct wait_ack_data_type wait_a_d[];
{
    char c1, c2, sys[MAX_CMD], f1[MAX_FILE_NAME], f2[MAX_FILE_NAME];

#if TRACE
printf("send_old_msg id=(%d) again to node=(%s)\n", wait_a_d[i].id, wait_a_d[i].record[0].to);
#endif
    switch(wait_a_d[i].type)
    {
        case MSG:
            c1 = WAIT; c2 = MSG; break;
```

```
            case DATA:
                c1 = DATA; c2 = DATA; break;
            default: break;
        } /* switch */
        sprintf(f1, "%c%d", c1, wait_a_d[i].id);
        sprintf(f2, "%c%d", c2, wait_a_d[i].id);
        sprintf(sys,"rcp %s %s:%s",f1,last_node(wait_a_d[i].record),f2);
        system(sys);
} /* send_old_msg */


/* ------------------  send_msg  ------------------- */
/* send message to other site */

send_msg(type, id, node)
char type;
int id;
char node[];
{
    char f1[MAX_FILE_NAME], f2[MAX_FILE_NAME], sys[MAX_CMD];
    int i;

    if (type == MSG)
    {
#if TRACE
printf("--------- send message MSG (%d) to node=(%s)\n", id, node);
#endif
        sprintf(f1, "w%d", id);
        sprintf(f2, "m%d", id);
    }
    else
    {
#if TRACE
printf("--------- send message DATA (%d) to node=(%s)\n", id, node);
#endif
        sprintf(f1, "d%d", id);
        sprintf(f2, "d%d", id);
    }
    sprintf(sys,"rcp %s %s:%s", f1, node, f2);
    system(sys);
} /* send_msg */


/* ------------------- get_time ------------------- */
/* get time stamp */

char *get_time()
{
    FILE *fopen(), *fp;
    char t[6][10], current[MAX_TIME_LEN], temp[MAX_CMD],
        file[MAX_FILE_NAME];
    int i;

    sprintf(file, "date%d", getpid());
```

```
    sprintf(temp, "date > %s", file);
    system(temp);
    fp = fopen(file, "r");
    for (i = 0; i < 6; i++)
        fscanf(fp, "%s", t[i]);
    i = atoi(t[2]);
    if (i < 10)
        sprintf(t[2], "0%d", i);
    sprintf(current, "%s-%s-%s-%s-%s-%s", t[0],t[1],t[2],t[3],t[4],t[5]);
    fclose(fp);
    unlink(file);
    return(current);
} /* get_time */


/* ---------- month_to_int ---------- */
/* convert month to an interger */

int month_to_int(t1, t2, t3)
char t1, t2, t3;
{
    char temp[3];

    sprintf(temp, "%c%c%c", t1, t2, t3);
    if (strcmp(temp, "Jan") == 0 ) return(1);
    if (strcmp(temp, "Feb") == 0 ) return(2);
    if (strcmp(temp, "Mar") == 0 ) return(3);
    if (strcmp(temp, "Apr") == 0 ) return(4);
    if (strcmp(temp, "May") == 0 ) return(5);
    if (strcmp(temp, "Jun") == 0 ) return(6);
    if (strcmp(temp, "Jul") == 0 ) return(7);
    if (strcmp(temp, "Aug") == 0 ) return(8);
    if (strcmp(temp, "Sep") == 0 ) return(9);
    if (strcmp(temp, "Oct") == 0 ) return(10);
    if (strcmp(temp, "Nov") == 0 ) return(11);
    if (strcmp(temp, "Dec") == 0 ) return(12);
    else return(0);
} /* month_to_int */


/* ---------- over_time ---------- */
/* if over time then return TRUE else return FALSE */

BOOLEAN over_time(table_time, current, interval)
char table_time[], current[];
int interval;
{
    int t1, t2;

    if (table_time[24] < current[24]) return(TRUE); /* years */
    if (table_time[24] > current[24]) return(FALSE);
    if (table_time[25] < current[25]) return(TRUE);
    if (table_time[25] > current[25]) return(FALSE);
    if (table_time[26] < current[26]) return(TRUE);
```

```
    if (table_time[26] > current[26]) return(FALSE);
    if (table_time[27] < current[27]) return(TRUE);
    if (table_time[27] > current[27]) return(FALSE);
    t1 = month_to_int(table_time[4], table_time[5], table_time[6]);
    t2 = month_to_int(current[4], current[5], current[6]);
    if (t1 < t2) return(TRUE);   /* month */
    if (t1 > t2) return(FALSE);
    if (table_time[8] < current[8]) return(TRUE); /* date */
    if (table_time[8] > current[8]) return(FALSE);
    if (table_time[9] < current[9]) return(TRUE);
    if (table_time[9] > current[9]) return(FALSE);
    if (table_time[11] < current[11]) return(TRUE); /* hours */
    if (table_time[11] > current[11]) return(FALSE);
    if (table_time[12] < current[12]) return(TRUE);
    if (table_time[12] > current[12]) return(FALSE);
    if (interval >= 10)  /* minutes */
    {
        interval = interval/10;
        if ((current[14] - table_time[14]) > interval)
            return(TRUE);
        else return(FALSE);
    }
    else
    {
        if (table_time[14] < current[14]) return(TRUE);
        if (table_time[14] > current[14]) return(FALSE);
        if ((current[15] - table_time[15]) > interval)
            return(TRUE);
        else return(FALSE);
    }
} /* over_time */


/* ---------------------  last_node  ---------------------- */
/* return the last node in the record */

char *last_node(record)
struct record_type record[];
{
    int i;

    for (i = 0; i < 10 && record[i].to[0] != MARK; i++)
        ;
    i--;
    if (i <10) return(record[i].to);
    else return("NONE");
} /* last_node */


/* ---------------------  check_time_out  ---------------------- */
/* check the wait list for timed-out message */

check_time_out(wait_a_d, ready_stack)
struct wait_ack_data_type wait_a_d[];
```

```
struct ready_type ready_stack[];
{
    int i, interval;
    char current[MAX_TIME_LEN];
    struct ack_type *ack;

    strcpy(current, get_time());
    for (i = 0; i < MAX_TIME_LIST; ++i)
    {
        switch(wait_a_d[i].wait_for)
        {
            case REQ: interval = wait_a_d[i].interval - 4;
                break;
            case DATA: interval = wait_a_d[i].interval;
                break;
            case ACK: interval = wait_a_d[i].interval + 50;
                break;
            default: break;
        }
        if (wait_a_d[i].id != END && (over_time(wait_a_d[i].time,current,interval)))
            if(wait_a_d[i].no_try <= MAX_TRY)
            { /* redo old path */
                wait_a_d[i].no_try++;
                strcpy(wait_a_d[i].time, current);
            /* only the last try time stamp will be kept */
                if (wait_a_d[i].wait_for == ACK)
                {
#if TRACE
printf("# message (%d) timed out for waiting ACK, resend message\n", wait_a_d[i].id);
#endif
                    send_old_msg(i, wait_a_d);
                }
                else
                {
#if TRACE
printf("# message (%d) timed out for DATA or REQ\n", wait_a_d[i].id);
#endif
                    ready_stack[ready_stack[0].id].wait_for = wait_a_d[i].wait_for;
                    ready_stack[ready_stack[0].id++].id = wait_a_d[i].id;
                }
            }
            else /* reroute the new path */
            {
#if TRACE
printf("# RE-ROUTE message (%d) for=(%c)\n",wait_a_d[i].id,wait_a_d[i].wait_for);
#endif
                if (wait_a_d[i].wait_for == ACK)
                {
                    ready_stack[ready_stack[0].id].wait_for = wait_a_d[i].wait_for;
                    ready_stack[ready_stack[0].id++].id = wait_a_d[i].id;
                    wait_a_d[i].id = END;
                }
                else
                {
```

```
                    ack = (struct ack_type *) malloc(sizeof(struct ack_type));
                    ack->id = wait_a_d[i].id;
                    ack->type = MSG;
                    ack->info = DEAD;
                    delete_wait_a_d(wait_a_d, ack);
                    free(ack);
                }
            }
        }
} /* time_out */


/* ---------- dump_record ---------- */
/* copy record 2 to record 1 */

dump_record(r1, r2)
struct record_type r1[], r2[];
{
    int i;

    for (i = 0; i < 10; i++)
    {
        strcpy(r1[i].to, r2[i].to);
        strcpy(r1[i].process, r2[i].process);
    }
} /* dump_record */


/* ---------- insert_wait_a_d ---------- */
/* insert message into wait list */

insert_wait_a_d(to, wait_a_d, id, type, p_record, wait_for, interval)
char to[];
struct wait_ack_data_type wait_a_d[];
int id;
char type;
struct record_type p_record[];
char wait_for;
int interval;
{
    int i;

    for (i = 0; i < MAX_TIME_LIST && id != END && id != wait_a_d[i].id; ++i) ;
    if (i >= MAX_TIME_LIST && id != END)
        for (i = 0; i < MAX_TIME_LIST && wait_a_d[i].id != END; ++i)
            ;
    if (i < MAX_TIME_LIST && id != END)
    {
        wait_a_d[i].id = id;
        wait_a_d[i].type = type;
        wait_a_d[i].wait_for = wait_for;
        strcpy(wait_a_d[i].time, get_time());
        if (wait_for == DATA || wait_for == REQ)
            wait_a_d[i].no_try++;
```

```
            else
                wait_a_d[i].no_try = 1;
            wait_a_d[i].interval = interval;
            dump_record(wait_a_d[i].record, p_record);
        }
} /* insert_wait_a_d */


/* ---------- delete_wait_a_d ---------- */
/* delete message from wait list */

delete_wait_a_d(wait_a_d, ack)
struct wait_ack_data_type wait_a_d[];
struct ack_type *ack;
{
    int i, j;
    char f1[MAX_FILE_NAME], f2[MAX_FILE_NAME], type, sys[MAX_CMD];
    struct messg_type *msg;
    BOOLEAN ok = TRUE;
    FILE *fopen(), *fp;

    for (i = 0; i < MAX_TIME_LIST &&
        (wait_a_d[i].id != ack->id ||
        wait_a_d[i].type != ack->type); i++)
        ;
    if (i < MAX_TIME_LIST)  /* found */
        wait_a_d[i].id = END;
    if (ack->info == DEAD || ack->info == DONE)
    {
        msg = (struct messg_type *) malloc(sizeof(struct messg_type));
        init_msg(msg, ack->id/1000);
        if (strcmp(host, msg->start_node) == 0)
        {
            if (ack->info == DEAD)
            {
                sprintf(f1, "dead%d",ack->id/100*100);
                fp = fopen(f1, "w");
                fprintf(fp, "Failed to continue job - %d", ack->id/100*100);
                fclose(fp);
            }
            else
            {
                ok = read_msg('w', ack->id, msg);
                sprintf(f1, "done%d",ack->id/100*100);
                write_msg_to_file(msg, f1);
            }
        }
        else
            send_ack(ack, msg->start_node);
        sprintf(f1, "w%d", ack->id/100);
        sprintf(f2, "d%d", ack->id/100);
        sprintf(sys, "rm %s*", f1);
        system(sys);
        sprintf(sys, "rm %s*", f2);
```

```
            system(sys);
            for (i = 0; i < MAX_TIME_LIST; i++)
                if (wait_a_d[i].id/100 == ack->id/100)
                    wait_a_d[i].id = END;
        }
        else
        {
            msg = (struct messg_type *) malloc(sizeof(struct messg_type));
            ok = read_msg('w', ack->id, msg);
            for (j = 0; ok && j < msg->next_h; ++j) /* clean up files */
            {
                sprintf(f1, "w%d", msg->history[j].id);
                sprintf(f2, "d%d", msg->history[j].id);
                unlink(f1);
                unlink(f2);
            }
            free(msg);
        }
    } /* delete_wait_a_d */


    /* ---------------      combine_info      ---------------- */
    /* combine messages */

    combine_info(t_m, tmp)
    struct messg_type *t_m, *tmp;
    {
        int i, k, temp, max_place;

        temp = pre_trans_no(tmp);
        if (t_m->id/100 == 1) max_place = MAX_PLACE1;
        else max_place = MAX_PLACE2;
        for (i = 0; i < max_place; i++)   /* combine marking */
            if (t_m->marking[i] == TRUE || tmp->marking[i] == TRUE)
                t_m->marking[i] = TRUE;
            else t_m->marking[i] = FALSE;

        t_m->inputs[1] = tmp->result; /* local information */

        for (i = t_m->next_h, k = 0; k < tmp->next_h; i++, k++) /* combine history */
        {
            t_m->history[i].id = tmp->history[k].id;
            t_m->history[i].trans = tmp->history[k].trans;
            t_m->history[i].op = tmp->history[k].op;
            strcpy(t_m->history[i].node, tmp->history[k].node);
            strcpy(t_m->history[i].time, tmp->history[k].time);
        }
        t_m->next_h = i;
    } /* combine_info */


    /* ----------------      before_merge      ---------------- */
    /* settings for merge transtion */
```

```
before_merge(trans_no, t_m, merge_node, resp)
int trans_no;
struct messg_type *t_m;
char merge_node[];
char *resp;
{
    int i, out_p, t1, mark, merge, b[MAX_PLACE2], times = 0;
    FILE *fp, *fopen();
    char f1[MAX_FILE_NAME], buf[30], c;
    struct messg_type *tmp;

    merge = t_m->net.t[trans_no].out_p[0];
    merge = t_m->net.p[merge].merge_trans;
    if (t_m->id % 100 != t_m->net.t[merge].in_p[0])
    {
        /* write marking and data to parent */
        sprintf(f1, "d%d", t_m->id);
        t_m->type = DATA;
        write_msg_to_file(t_m, f1);
        /* send data to merge node */
        if (strcmp(merge_node, host) != 0)
            send_msg(DATA, t_m->id, merge_node);
        *resp = 'c'; /* c - child */
    }
    else /* master copy */
    {
        /* if the current node is not the merge node, then send out the msg to merge node */
        wait(&status);
        if (strcmp(merge_node, host) == 0)
        {
            t1 = t_m->id / 100 * 100 + t_m->net.t[merge].in_p[1];
            for(fp = NULL, times = 1; fp == NULL && ++times <= 3;)
            {
                sprintf(f1, "d%d", t1);
                fp = fopen(f1, "r");
                if (fp == NULL) sleep(3);
            }
            fclose(fp);
            /* before real merge action, we have to update the marking first, up to this point, only have
               to know the marking of the child */
            if (times <= 3)
            {
                tmp = (struct messg_type *) malloc(sizeof(struct messg_type));
                init_msg(tmp, t_m->id/1000);
                read_msg(DATA, t1, tmp);
                combine_info(t_m, tmp);
                *resp = 'o'; /* o - ok */
                free(tmp);
            }
            else
            {
                *resp = 'f';  /* f - failed */
                t_m->marking[t_m->net.t[merge].in_p[1]] = 1;
                t_m->inputs[1] = -999;
```

```
            }
        }
        else /* write out the msg to a file and send it to the merge node */
        {
            sprintf(f1, "w%d", t_m->id);
            write_msg_to_file(t_m, f1);
            send_msg(MSG, t_m->id, merge_node);
            *resp = 'm';  /* m - message */
        }
    }
} /* before_merge */


/* ----------------  copy_msg  ---------------- */
/* copy message 2 to message 1 */

copy_msg(m1, m2)
struct messg_type *m1, *m2;
{
    int i, max_place, max_data;

    m1->id = m2->id;
    m1->type = m2->type;
    m1->next_h = m2->next_h;
    if (m2->id/1000 == 1)
        max_place = MAX_PLACE1;
    else
        max_place = MAX_PLACE2;
    for (i = 0; i < max_place; i++)
        m1->marking[i] = m2->marking[i];
    m1->result = m2->result;
    m1->inputs[0] = m2->inputs[0];
    m1->inputs[1] = m2->inputs[1];
    for (i = 0; i < m2->next_h; i++)
    {
        m1->history[i].id = m2->history[i].id;
        m1->history[i].trans = m2->history[i].trans;
        m1->history[i].op = m2->history[i].op;
        strcpy(m1->history[i].node, m2->history[i].node);
        strcpy(m1->history[i].time, m2->history[i].time);
    }
}


/* ----------------  update_marking  ---------------- */
/* update the marking after a transtion is completed */

update_marking(trans_no, t_m)
int trans_no;
struct messg_type *t_m;
{
    int i;

    /* fine the out_put places according to trans_no update the marking for all input and output places */
```

```
    for (i = 0; t_m->net.t[trans_no].in_p[i] != END; i++)
        t_m->marking[t_m->net.t[trans_no].in_p[i]]
        = FALSE;
    for (i = 0; t_m->net.t[trans_no].out_p[i] != END; i++)
        t_m->marking[t_m->net.t[trans_no].out_p[i]]
        = TRUE;
}


/* ---------------  update_msg  ---------------- */
/* update the message after a transtion is completed */

update_msg(trans_no, t_m, msg, resp, var)
int trans_no;
struct messg_type *t_m, *msg;
char *resp;
struct data_type var[];
{
    int index1, index2;
    char var1[2], var2[2], merge_node[MAX_NODE_NAME], f1[6], f2[6];

    t_m->inputs[0] = -999; t_m->inputs[1] = -999;
    strcpy(merge_node,t_m->net.p[t_m->net.t[trans_no].out_p[0]].merge_next);
    if (t_m->net.t[trans_no].act.op != 'f')
    {
        update_marking(trans_no, t_m);
        t_m->id = t_m->id / 100 * 100 + t_m->net.t[trans_no].out_p[0];
    }
    t_m->history[t_m->next_h].id = t_m->id;
    t_m->history[t_m->next_h].trans = trans_no;
    t_m->history[t_m->next_h].op = t_m->net.t[trans_no].act.op;
    strcpy(t_m->history[t_m->next_h].node, host);
    strcpy(t_m->history[t_m->next_h++].time, get_time());
    send_ack_cleanup(t_m);
    if (strcmp(merge_node, "NO") != 0) /* before merge transition */
        before_merge(trans_no, t_m, merge_node, resp);
    copy_msg(msg, t_m);
    sprintf(f2, "w%d", t_m->id);
    write_msg_to_file(t_m, f2);
} /* update_msg */
```

JOB EXECUTION
IN A DISTRIBUTED ENVIRONMENT
USING PETRI NETS

by

LING-LING HSU
B.S., Fu-Jen University, 1982

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

The concept of distributed computing is in broad use today and will become even more popular in the future. All resource-sharing systems involving two or more processors are considered distributed systems. A system for automatic routing of jobs is needed in a distributed environment to ease the task of users and to ensure the correctness of results. A system has been constructed which contains three major components: Control nets (CNs), intelligent tokens, and Control net agents. Routing and synchronization requirements of a job are represented via an extended Petri net. Predicates are used in conjunction with transitions of this Petri net. Electronic versions of jobs are designed as intelligent tokens which contain both the control and data information that are needed to perform jobs. A Control net and its marking supply control information as well as contraints on how the job's data may be manipulated. A process called a Control net agent exists at every site in a distributed environment and is responsible for interpreting the CN, performing functions, and physical routing of jobs. This prototype system was implemented to show that execution of jobs can be controlled through the use of a CN.